Streszczenie

Niniejsza praca ma na celu przedstawienie i omówienie sposobu pozwalającego na znajdowanie pierwiastków rzeczywistych wielomianu w określonym przedziale. Najważniejszym aspektem był dobór odpowiedniej metody tak, by była skuteczna w każdym przypadku, a nie tylko w określonych warunkach. Przykładem takiego sposobu jest wykorzystanie twierdzenia Sturma. Przed jego zastosowaniem należy dokonać eliminacji pierwiastków wielokrotnych wielomianu, by móc następnie zdefiniować ciąg Sturma. Na jego podstawie jesteśmy w stanie określić liczbę różnych pierwiastków w dowolnym przedziale. Na mocy twierdzenia jest ona równa różnicy liczby zmian znaku na obu jego krańcach. Badany przedział możemy więc dowolnie zmniejszać, by z wymaganą precyzją wskazać wartość szukanych pierwiastków.

Ważnym aspektem pracy jest aplikacja konsolowa w języku C++, pozwalająca na znajdowanie pierwiastków wprowadzonego przez użytkownika wielomianu. Do jej wykonania niezbędna jest implementacja działań na wielomianach oraz obsługa dużych wartości liczbowych o wysokiej precyzji. W tym celu program korzysta z biblioteki MPIR. By zapewnić odpowiednią jakość, powstał osobny moduł aplikacji, w którym zostały zdefiniowane odpowiednie testy funkcjonalne programu.

Ciekawym zagadnieniem pracy jest porównanie różnych typów struktur, pozwalających na reprezentację wielomianu. Pierwszą z nich jest tablica, w której trzymane są wszystkie współczynniki wielomianu. Drugą zaś jest mapa, jako przykład struktury, o braku bezpośredniego dostępu do wszystkich jej elementów, co pozwala na posiadanie informacji wyłącznie o niezerowych współczynnikach. Przeprowadzone zostały testy porównujące wydajność działania obu struktur. Wyniki obu z nich zostały porównane zarówno z wartościami teoretycznymi, jak i ze sobą. Dodatkowo została przeprowadzona analiza, która ze strukturach jest lepsza dla różnych typów wielomianów.

Słowa kluczowe:

wielomian, pierwiastki wielomianu, twierdzenie Sturma, biblioteka mpir

Dziedzina nauki i techniki, zgodnie z wymogami OECD:

Nauki przyrodnicze, Matematyka, Nauki o komputerach i informatyka

Abstract

The purpose of this thesis is presentation and discussion about the way allowing finding real roots of the polynomial in examined range. The most important aspect was a choice of the appropriate method to be effective in each case and not only under certain conditions. An example of such way is the use of Sturm's theorem. Before using it, it is necessary to eliminate of multiple roots of the polynomial, then be able to define the Sturm's sequence. Based on it, we are able to determine the number of distinct real roots located in an interval. By theorem it is equal to the difference changes sign at both ends. The test interval can be so arbitrarily reduce to indicate the value of searched roots with the required precision.

The important aspect of this thesis is a console application, made in C++, allowing finding roots of the polynomial, which is entered by a user. For its implementation it is necessary to implement actions on polynomials and handling large numbers of high-value precision. For this purpose, the program uses the MPIR library. To ensure the quality of the program, was created a separate application module, which includes the relevant functional tests.

An interesting issue is to compare the different types of structures, allowing a polynomial representation. The first of them is the array, which are kept all the coefficients of the polynomial. Another one is a map, as an example of structure with a lack of direct access to all of its elements, which allows you to have information only about the non-zero coefficients. They were performed tests comparing the operating efficiency of both structures. The results of both of them are compared with both the theoretical values as well as with each other. In addition, the analysis was conducted, which of the structures is better for different types of polynomials.

Keywords:

polynomial, roots of polynomial, Sturm's theorem, mpir library

Field of science and technology in accordance with OECD requirements:

Natural sciences, Mathematics, Computer and information sciences

Spis treści

Rozdział 1

Wstęp

Celem niniejszej pracy jest implementacja algorytmu, pozwalającego na obliczanie pierwiastków rzeczywistych dowolnego wielomianu. Jednym z najlepszych sposobów na zrobienie tego jest skorzystanie z twierdzenia Sturma. Planuję zbadać, jaki sposób reprezentacji wielomianu jest najbardziej wydajny. Dodatkowo zamierzam także porównać możliwości stworzonej przeze mnie aplikacji z innymi programami.

Zacznę od przedstawienia tego, czym są wielomiany. Zapoznanie się bowiem z podstawowymi definicjami i twierdzeniami jest moim zdaniem niezbędne, by dobrze zrozumieć badany problem i móc wypracować jego skuteczne rozwiązanie. Warte dość szczegółowego opisu są działania charakterystyczne dla wielomianów. Są one analogiczne jak w przypadku liczb. Należą do nich dodawanie, odejmowanie, mnożenie, dzielenie, obliczanie reszty z dzielenia. Dwa ostatnie są operacjami najbardziej skomplikowanymi i tym samym najbardziej złożonymi pod względem czasowym. Dodatkowo warto zaznaczyć, że w przypadku wielomianu możliwe jest obliczanie pochodnej, największego wspólnego dzielnika oraz najmniejszej wspólnej wielokrotności.

Jak wspomniałem na początku, w pracy wykorzystam metodę ciągu Sturma. Pozwala ona znaleźć wszystkie pierwiastki rzeczywiste danego wielomianu, występujące w podanym przedziale. Zakłada ona zbudowanie ciągu wielomianów, coraz niższych stopni, a następnie porównywanie ze sobą ich wartości w danym punkcie. Na tej podstawie zostaje obliczona liczba zmian znaku w ciągu Sturma. Różnica tych wartości w obu krańcach badanego przedziału jest liczbą pierwiastków rzeczywistych, które w nim występują. W przeciwieństwie do wielu innym metod, twierdzenie Sturma jest skuteczne dla każdego wielomianu. Jedynym ograniczeniem jest konieczność dokonania eliminacji pierwiastków wielokrotnych danego wielomianu. By to uczynić, wystarczy podzielić go przez największy wspólny dzielnik tego wielomianu oraz jego pochodną.

W przypadku obliczeń na wielomianach ważnym aspektem są obliczenia na bardzo dużych liczbach z dowolną precyzją. Konieczne jest to zwłaszcza, gdy mamy do czynienia z wielomianami wysokich stopni. Wówczas każda liczba, której wartość bezwzględna jest większa od jedynki, po podniesieniu do wysokiej potęgi, jest bardzo duża. Dla przykładu wielomian x^{100} w punkcie $x_1 = 10$ ma wartość 10^{100}. Z kolei w przypadku pozostałych liczb, wartość ta bardzo szybko zbliża się do

zera, np. w punkcie $x_2 = 0.1$ wartość tego wielomianu jest równa zaledwie $(0.1)^{100}$. Aplikacja jest implementowana w języku C++, a żaden z wbudowanych w niego typów nie posiada możliwości przechowywania, aż tak dokładnych i na tyle dużych wartości. Rozwiązaniem jest tutaj skorzystanie z biblioteki dużych liczb, z dowolną precyzji – mpir. Posiada ona swoją implementację dla wielu języków programowania, w tym C i C++. Warto zaznaczyć, że rozwiązanie to jest bardzo wydajne, ponieważ najbardziej krytyczne funkcje zostały zaimplementowane w asemblerze, a cała biblioteka jest optymalizowana pod kątem użycia na konkretnym procesorze. Projekt zakłada istnienie trzech głównych modułów. Pierwszym z nich jest biblioteka statyczna, w której zawarte będą wszystkie niezbędne klasy i funkcje. Drugim jest aplikacja konsolowa, która pozwoli na interakcję użytkownika programu z biblioteką i wykonywaniem jej metod dla konkretnych wielomianów. Zaś ostatnim są testy funkcjonalne, pozwalające na przetestowanie funkcjonalności programu, w celu zapewniania mu odpowiedniej jakości i łatwego monitorowania jego poprawności. Przewiduję implementację wielomianów, bazującą na dwóch odmiennych strukturach. Pierwszą z nich jest tablica, która wymusza reprezentację wszystkich jego współczynników, także zerowych. Druga koncepcja zakłada przechowywanie informacji wyłącznie o niezerowych wartościach współczynników wielomianów. Opierać się ona będzie na drzewiastej strukturze, w której brak jest bezpośredniego dostępu do dowolnego elementu. Do reprezentacji powyższych struktur planuję wykorzystać typy z biblioteki SDL, jakimi są wektor oraz mapa.

W dalszej części pracy zawarty będzie opis wszystkich stworzonych funkcji i klas, w tym dokładna analiza porównawcza obu typów wielomianów. Planowane jest stworzenie abstrakcyjnej klasy, posiadającej metody, które będą całkowicie niezależne od reprezentującej dany wielomian struktury. Wszystkie pozostałe metody będą w niej zadeklarowane i oznaczone jako czysto wirtualne. Zgodnie z paradygmatami programowania obiektowego wymagane będzie przeciążenie każdej z nich by móc stworzyć instancję klasy danego wielomianu. Poprzez zdefiniowanie odpowiedniego interfejsu, łatwiejsza będzie także potencjalna rozbudowana aplikacji o ewentualne zdefiniowanie kolejnych typów wielomianów, bazujących na innych niż wymienione wyżej struktury.

Do programu zostanie dołączona dokładna instrukcja programu oraz niezbędne informację o wymaganym środowisku. Warto zaznaczyć, że celem projektu jest stworzenie aplikacji działającej pod systemem Windows. Nie jest natomiast planowane jej uruchomienie pod Linuksem. Należy także zaznaczyć, że wykorzystywana biblioteka mpir będzie skompilowana dla procesorów Intel i7 i wykorzystanie aplikacji używając innego sprzętu może być niemożliwe lub wymagać wykonania dodatkowych czynności. Jak zostało wspomniane wcześniej, projekt będzie posiadał moduł pozwalający na zdefiniowanie i wykonywanie testów funkcjonalnych. Rozwiązaniem, które to umożliwia, jest framework testów jednostkowych, wbudowany w narzędzie deweloperskie Microsoft Visual Studio 2015. Napisane testy będą dotyczyć zarówno krytycznych funkcji programu, jak i weryfikować poprawne działanie całego projektu.

Zostaną przeprowadzone testy wydajnościowe, które porównają działanie czasowe obu typów wielomianów. Rozpatrzone będą specyficzne przypadki testowe, faworyzujące poszczególne reprezentacje oraz testy dla losowo dobranych współczynników wielomianów. Nastąpi analiza, na podstawie której, zostaną wyciągnięte wnioski, w jakich przypadkach każda z testowanych struktur jest bardziej wydajna. Otrzymane wyniki zostaną porównane z wartościami teoretycznymi dla

poszczególnych typów wielomianów. Dodatkowo możliwości programu zostaną porównane z inną aplikacją, pozwalającą znajdować pierwiastki rzeczywiste wielomianów.

Rozdział 2

Wprowadzenie

2.1 Wielomiany

2.1.1 Definicja

Definicja 1
Wielomianem zmiennej rzeczywistej x nazywamy wyrażenie:
$W(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + ... + a_{n-1}x + a_n$,
gdzie $a_0, a_1, a_2, ..., a_{n-1}, a_n \in R$ *i* $n \in N$

Liczby $a_0, a_1, a_2, ..., a_{n-1}, a_n$ nazywamy współczynnikami wielomianu, natomiast n nazywamy stopniem wielomianu.

Szczególnym przypadkiem wielomianu jest jednomian.

Definicja 2
Jednomianem zmiennej rzeczywistej x nazywamy wielomian zmiennej rzeczywistej x, który posiada co najwyżej jeden wyraz niezerowy.

Można, więc rozumieć wielomian jako skończoną sumę jednomianów. Jednomianem stopnia zerowego jest stała, pojedyncza liczba rzeczywista, która w szczególności może być zerem.

Definicja 3
Wielomianem zerowym nazywamy wielomian wyrażony wzorem: $W(x) = 0$.

W dalszej części, jeżeli nie zaznaczymy inaczej, mówiąc wielomian, będziemy mieli na myśli pewien wielomian, nie będący wielomianem zerowym.

2.1.2 Podstawowe działania na wielomianach

Na wielomianach, tak jak na liczbach, możemy wykonywać podstawowe działania. Należą do nich: porównywanie, dodawanie, odejmowanie, mnożenie, dzielenie, a także obliczanie reszty z dzielenia.

Jako, że wielomian zmiennej x możemy traktować jak funkcję jednej zmiennej, możemy także policzyć z niego pochodną.

Porównywanie wielomianów

Porównywanie należy do najbardziej elementarnych działań na wielomianach. Wymaga ono zwykłego porównania kolejnych współczynników, a jego długość trwania, zależy od ich liczby. Zapoznajmy się z twierdzeniem dotyczącym operacji porównywania wielomianów.

Twierdzenie 1
Dwa wielomiany uważamy za równe wtedy i tylko wtedy, gdy są tego samego stopnia, a ich kolejne współczynniki są równe.

Powyższe twierdzenie nie jest złożone, nie mniej w celu pełnego zrozumienia, zilustrujmy je przykładem.

Przykład 1
Mamy dane wielomian W_1 *oraz wielomian* W_2.
$W_1(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$
$W_2(x) = b_0x^n + b_1x^{n-1} + ... + b_{n-1}x + b_n$
Wielomiany W_1 *oraz* W_2 *są równe wtedy i tylko wtedy gdy* $\forall i \in N \ a_i = b_i$.

Można zauważyć potencjalny wpływ reprezentacji wielomianu na szybkość operacji porównania. Gdy mamy do czynienia z wielomianem, w którym uwzględniamy każdy współczynnik, także gdy jest on zerowy, złożoność czasowa porównania jest liniowa względem stopnia wielomianu. Natomiast w przypadku, gdy pomijamy wszystkie zerowe współczynniki wielomianu, złożoność również jest liniowa, ale tym razem względem liczby niezerowych współczynników wielomianów. Jak widać, w sytuacji, gdy stopień wielomianu jest znacznie większy od liczby zerowych współczynników, reprezentacja wielomianu ma niebagatelne znaczenie.

Dodatkowo, podobnie jak w przypadku porównywania liczb i sprawdzania kolejnych bitów, operacja porównania kończy się w momencie stwierdzenia, że porównywane współczynniki są różne lub porównaliśmy ze sobą już wszystkie współczynniki. Wynika z tego, że zakładając stały czas porównywania dwóch liczb, będących współczynnikami wielomianów, operacja porównania różnych wielomianów nigdy nie jest dłuższa od stwierdzenia, że porównywane wielomiany są równe.

Suma wielomianów

Dodawanie to kolejne elementarne działanie na wielomianach, które nie wymaga wykonywania skomplikowanych obliczeń.

Twierdzenie 2
Aby dodać dwa wielomiany, należy dodać ich współczynniki, znajdujące się przy tych samych potęgach.

Podobnie jak w przypadku porównywania czas dodawania wielomianów jest liniowy, a ich reprezentacja ma zasadniczy wpływ na liczbę operacji dodawania. Pokażmy zastosowanie powyższego twierdzenia na przykładzie.

Przykład 2
Mamy dane wielomian W_1 *oraz wielomian* W_2*.*
$W_1(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$
$W_2(x) = b_0x^n + b_1x^{n-1} + ... + b_{n-1}x + b_n$
Zdefiniujmy trzeci wielomian:
$W_3(x) = W_1(x) + W_2(x)$.
Wówczas:
$W_3(x) = (a_0 + b_0)x^n + (a_1 + b_1)x^{n-1} + ... + (a_{n-1} + b_{n-1})x + (a_n + b_n)$

Na powyższym przykładzie łatwo zaobserwować, że stopień sumy dwóch wielomianów nie może być większy od większego ze stopni dodawanych wielomianów. Znajduje to potwierdzenie w twierdzeniu, dotyczącym stopnia sumy wielomianów.

Twierdzenie 3
$\deg(W_1 + W_2) \leqslant \max(\deg(W_1), \deg(W_2))$

W przedstawionym twierdzeniu, należy zwrócić uwagę na operator mniejsze równe. W przypadku, gdy oba te wielomiany są tego samego stopnia, o przeciwnym współczynniku przy najwyższej potędze, to stopień ten będzie mniejszy.

Różnica wielomianów

Odejmowanie to operacja bliźniacza do dodawania, nie tylko w przypadku liczb, ale także w przypadku wielomianów. By pokazać olbrzymie podobieństwo tych operacji, zacznijmy od zapoznania się z definicją wielomianu przeciwnego.

Definicja 4
Wielomianem przeciwnym nazywamy wielomian, którego wszystkie współczynniki są przeciwne do danych.

Spójrzmy na poniższy przykład, pokazujący, że dla każdego wielomianu można bardzo prosto zdefiniować wielomian przeciwny, zmieniając znak wszystkich jego współczynników.

Przykład 3
Mamy dany wielomian W_1*.*
$W_1(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$
Zdefiniujmy drugi wielomian: $W_2(x) = -W_1(x)$*. Wówczas:*
$W_1(x) = -a_0x^n + (-a_1)x^{n-1} + ... + (-a_{n-1})x + (-a_n)$

Wiemy już, czym jest wielomian przeciwny. Przedstawmy teraz twierdzenie mówiące jak odejmować od siebie wielomiany.

Twierdzenie 4
Aby obliczyć różnicę wielomianów W_1 i W_2, należy dodać ze sobą wielomiany W_1 i $-W_2$, czyli wielomian przeciwny do wielomianu W_2.

Jak widać, przedstawione twierdzenie potwierdza analogię obliczania różnicy i sumy wielomianów. Spójrzmy na przykład pokazujący, jak obliczać różnicę wielomianów, potrafiąc już je do siebie dodawać.

Przykład 4
Mamy dane wielomian W_1 oraz wielomian W_2. $W_1(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$
$W_2(x) = b_0x^n + b_1x^{n-1} + ... + b_{n-1}x + b_n$
Zdefiniujmy trzeci wielomian: $W_3(x) = W_1(x) - W_2(x)$. *Wówczas:*
$W_3(x) = (a_0 - b_0)x^n + (a_1 - b_1)x^{n-1} + ... + (a_{n-1} - b_{n-1})x + a_n - b_n$

Warto zauważyć, że wielomianem neutralnym ze względu na dodawanie i odejmowanie jest wielomian $W(x) = 0$. Oznacza to, że po dodaniu lub odjęciu wielomianu neutralnego, dostaniemy wynik, będący danym wielomianem.

Iloczyn wielomianów

Mnożenie to kolejna operacja zaliczająca się do podstawowych działań na wielomianach. Jego zasady przypominają nieco zwykłe mnożenie. Dokonujemy przemnożenia odpowiednich wyrazów, z tą różnicą, że w tym przypadku po prostu dodajemy wartości potęg dla odpowiednich współczynników. Zapoznajmy się z twierdzeniem, mówiącym dokładnie jak należy obliczać iloczyn wielomianów.

Twierdzenie 5
Aby pomnożyć dwa wielomiany, należy wymnożyć przez siebie wyrazy obu wielomianów, a następnie dodać do siebie wyrazy podobne.

Jak wynika z przedstawionej definicji poza mnożeniem dwóch liczb, mnożenie wielomianów w części polega na redukcji wyrazów podobnych, czyli operacji bazującej na dodawaniu. Spójrzmy na przykład, pokazujący jak definiuje się wielomian, będacy iloczynem dwóch wielomianów.

Przykład 5
Mamy dane wielomian W_1 oraz wielomian W_2.
$W_1(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$
$W_2(x) = b_0x^m + b_1x^{m-1} + ... + b_{m-1}x + b_m$
Zdefiniujmy trzeci wielomian: $W_3(x) = W_1(x) \cdot W_2(x)$. *Wówczas:*
$W_3(x) = (a_0b_0)x^{n+m} + (a_0b_1 + a_1b_0)x^{n+m-1} + (a_0b_2 + a_1b_1 + a_2b_0)x^{n+m-1} + ... + (a_{n-2}b_m + a_{n-1}b_{m-1} + a_nb_{m-2})x^2 + (a_{n-1}b_m + a_nb_{m-1})x + a_nb_m$

Można zauważyć, że po wymnożeniu wszystkich współczynników wielomianu liczba wyrazów iloczynu wynosi $(n+1)(m+1)$. Po dokonaniu redukcji wyrazów podobnych, liczba ta ulega zmniejszeniu do wartości $n + m + 1$. Oznacza to zmianę liczby wyrazów z wartości kwadratowej, do wartości liniowej względem stopni wielomianów. Liczba wyrazów podobnych, po przemnożeniu dwóch wielomianów jest symetryczna względem wykładników potęg poszczególnych współczynników. Można zauważyć, że skrajne wyrazy posiadają tylko po jednym wyrazie potęgi, a zbliżając się do współczynników o środkowych indeksach, liczba ta wzrasta, aż do wartości równej połowie stopnia otrzymanego wielomianu.

Widzimy, że czas operacji mnożenia wielomianów jest kwadratowy, względem stopni mnożonych przez siebie czynników. Należy zauważyć, że jeżeli użyjemy reprezentacji wielomianu, w których posiadamy informację tylko o jego niezerowych współczynnikach, to czas operacji mnożenia będzie nadal kwadratowy, ale względem liczby tych współczynników. Dla wielomianów wysokich stopni, w których zaledwie kilka współczynników jest niezerowych różnica ta może być niebagatelna i w skrajnych przypadkach czas operacji może zmniejszyć się z kwadratowego, do czasu stałego.

Na podstawie powyższego przykładu, możemy także zaobserwować, że stopień wielomianu, będącego iloczynem dwóch wielomianów niezerowych, jest standardowo równy sumie stopni tych wielomianów. Wyjątkiem jest sytuacja, gdy jeden z czynników jest wielomianem zerowym. Wówczas wynik takiej operacji również będzie wielomianem zerowym. Fakt ten znajduje potwierdzenie w poniższym twierdzeniu.

Twierdzenie 6
$\deg(W_1 \cdot W_2) = \deg(W_1) + \deg(W_2)$, *dla* $W_1(x) \neq 0, W_2(x) \neq 0$
$W_3(x) = W_1(x) \cdot W_2(x) = 0$, *w pozostałych przypadkach*

Z powyższego twierdzenia można zauważyć, że stopień otrzymanego wielomianu nigdy nie będzie wyższy od dwukrotności większego ze stopni mnożonych wielomianów.

Iloraz wielomianów

Dzielenie to zdecydowanie najtrudniejsza z elementarnych operacji na wielomianach. Aby dobrze zrozumieć jego zasady zapoznajmy się z definicją podzielności wielomianów oraz dzielnika wielomianu.

Definicja 5
Wielomian $W(x)$ *nazywamy podzielnym przez niezerowy wielomian* $P(x)$ *wtedy i tylko wtedy, gdy istnieje taki wielomian Q(x), że spełniony jest warunek* $W(x) = P(x) \cdot Q(x)$. *Wówczas: wielomian* $Q(x)$ *nazywamy ilorazem wielomianu* $W(x)$ *przez* $P(x)$, *zaś wielomian* $P(x)$ *nazywamy dzielnikiem wielomianu* $W(x)$.

Bardzo ważnym aspektem obliczania ilorazu wielomianów jest reszta z dzielenia. Spójrzmy na poniższą definicję.

Definicja 6
Wielomian $R(x)$ nazywamy resztą z dzielenia wielomianu $W(x)$ przez niezerowy wielomian $P(x)$ wtedy i tylko wtedy, gdy istnieje taki wielomian $Q(x)$, że spełniony jest warunek $W(x) = P(x) \cdot Q(x) + R(x)$ i $\deg R(x) < \deg(P)$.

Łatwo zauważyć analogię w wyżej przedstawionych wzorach. Różnią się one właśnie wielomianem $R(x)$, czyli resztą z dzielenia. Gdy jest ona wielomianem zerowym, to znaczy, że mamy do czynienia z dzieleniem bez reszty i mówimy o podzielności dwóch wielomianów. Spórzmy na przykład, w którym zdefiniowane zostały dwa wielomiany, będące ilorazem i resztą z dzielenia dwóch wielomianów.

Przykład 6
Mamy dane wielomian W oraz wielomian P.
$W(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$
$P(x) = b_0x^m + b_1x^{m-1} + ... + b_{m-1}x + b_m$
Niech $Q(x)$ jest ilorazem wielomanów $W(x)$ i $P(x)$ oraz $R(x) = W(x) \bmod P(x)$. Wówczas:
$Q(x) = c_0x^{n-m} + c_1x^{n-m-1} + ... + c_{n-m-1}x + c_{n-m}$, *gdzie* $c_0 \neq 0$
$R(x) = d_0x^{n-m-1} + d_1x^{n-m-2} + ... + d_{n-m-2}x + d_{n-m-1}$

Zwróćmy uwagę na potęgi stojące przy najwyższych potęgach wielomianów $Q(x)$ oraz $R(x)$. Widzimy, że stopień ilorazu wielomianów jest zawsze równy różnicy stopni wielomianów, będących dzielną i dzielnikiem. Najważniejszym aspektem jest jednak fakt, że stopień reszty z dzielenia wielomianów jest zawsze mniejszy od stopnia ilorazu. Nie można natomiast ustalić jego wartości, bez dokładnej znajomości wielomianów W i P. W przykładzie podkreślony został fakt, że współczynnik stojący przy x, o potędze $n - m - 1$ może być zerem. To samo tyczy się także kolejnych współczynników. Gdy wszystkie one są zerami, to znaczy, że mamy do czynienia z resztą, będącą wielomianem zerowym. Oznacza to wówczas, że wielomian W jest podzielny przez wielomian P. Poniżej znajduje się twierdzenie, o stopniach wielomianów, będących ilorazem i resztą z dzielenia.

Twierdzenie 7
$\deg(W_1 \bmod W_2) < \deg(W_1/W_2) = \deg(W_1) - \deg(W_2)$

Jak widać, twierdzenie potwierdza nasze obserwacje i wnioski dotyczące stopni obu wielomianów.

Pochodna wielomianu

Wielomiany jako przykład funkcji ciągłej, pozwalają na obliczanie pochodnych. By przekonać się, że jest to przykład jednej z prostszych operacji na wielomianach, zapoznajmy się z definicją.

Definicja 7
Dany jest wielomian W, określony wzorem $W(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$, gdzie $n \geqslant 1$. Pochodną wielomianu W nazywamy wielomian W' i wyrażamy wzorem: $W'(x) = na_0x^{n-1} + (n-1)a_1x^{n-2} + ... + 2a_{n-2}x + a_{n-1}$

Widzimy, że powstały wielomian powstał poprzez pomnożenie wartości każdego z współczynników przez stojącą przy danym wyrazie potęgę, a następnie obniżenie jej wartości o jeden. W ten sposób potęgi wszystkich wyrazów wielomianu obniżają się. Wyjątkiem jest tutaj potęga o wartości zero, czyli stała. Pochodna z funkcji stałej jest zawsze równa zero, dlatego została pominięta w powyższym wzorze. O stopniu pochodnej wielomianu mówi poniższe twierdzenie.

Twierdzenie 8
$\deg(W') = \deg(W) - 1, dla\ \deg(W) > 0$
$W(x) = 0$, *w pozostałych przypadkach*

Jak widać dla wszystkich wielomianów, nie będących stałą liczbową, stopień ich pochodnej ulega zmniejszeniu o jeden. Czas operacji obliczania pochodnej wielomianu jest porównywalny, z obliczaniem sumy wielomianów, gdyż wystarczy, że dokonamy jednokrotnego obliczania każdego z współczynników.

2.1.3 Największy wspólny dzielnik wielomianów

Zacznijmy od zastanowienia się, co znaczy, że dany wielomian dzieli inny wielomian. Zapoznajmy się z poniższą definicją.

Definicja 8
Wspólnym dzielnikiem dwóch wielomianów W_1 i W_2, nazywy taki wielomian W_3, że gdy podzielimy dowolny z tych dwóch wielomianów przez W_3 to otrzymamy zerową resztę z dzielenia.

By zapewnić jednoznaczność wielomianów, w dalszej części mówiąc o NWD dwóch wielomianów zawsze będziemy mieli na myśli taki wielomian, którego współczynnik stojący przy najwyższej potędze jest równy 1. Zauważmy, że powyższa definicja jest analogiczna, jak w przypadku liczb naturalnych. Co oznacza największy wspólny dzielnik? W przypadku liczb jest to dowolna liczba naturalna spełniająca powyższy warunek. Podobnie jest dla wielomianów. Spośród wszystkich wielomianów, które spełniają to kryterium, jest to ten, którego stopień jest najwyższy. Potwierdza to poniższa definicja.

Definicja 9
Największym wspólnym dzielnikiem dwóch wielomianów W_1 i W_2, nazywy ten wspólny dzielnik, którego stopień jest najwyższy.

Również ta definicja jest analogiczna, jak w przypadku liczb, gdzie szukamy tego dzielnika, który jest największy.

Zapoznajmy się teraz z twierdzeniem dotyczącym sposobu, w jaki obliczamy nwd dwóch wielomianów. Metoda ta jest również znana pod pojęciem algorytmu Euklidesa.

Twierdzenie 9
Aby obliczyć największy wspólny dzielnik dla wielomianów W_1 i W_2, musimy dokonać obliczenia

kolejnych reszt z dzielenia. Pierwsza z nich jest obliczana dla ilorazu W_1 i W_2. Kolejne obliczane są dla dotychczasowego dzielnika i otrzymanej reszty. Dzielenia kończą się w momencie, gdy otrzymana reszta jest stopnia zerowego. Jeżeli jest ona wielomianem zerowym, oznacza to, że największym wspólnym dzielnikiem jest ostatni dzielnik. W przeciwnym wypadku, czyli w momencie, gdy wielomian ten jest liczbą różną od zera, najwiekszy wspólny dzielnik jest wielomianem stopnia zerowego, a oba wielomiany nie mają wspólnych pierwiastków.

Zauważmy, że na podstawie twierdzenia, przedstawionego wcześniej, wiemy, że reszta z dzielenia dwóch wielomianów jest mniejsza od różnicy ich stopni. Oznacza, to że jej stopień będzie maleć w przypadku kolejnych dzieleń. Fakt te, daje nam gwarancję, że liczba dzieleń, których musimy dokonać jest skończona i nie większa niż stopień wielomianu W_2. Przeanalizujmy poniższy przykład.

Przykład 7

Mamy dane wielomiany: $W_1(x) = x^2(x-1)(x-2)^2$ oraz $W_2(x) = (x-1)(x-2)(x-3)$. Gdy mamy wielomiany rozłożone na czynniki, wartość $NWD(W_1, W_2)$ jest oczywista i równa iloczynowi wspólnych czynników. Zatem w tym przypadku wynosi ona $(x-1)(x-2)$. Zapisując otrzymany wielomian w innej postaci otrzymamy $x^2 - 3x + 2$. Wymnóżmy teraz kolejne czynniki wielomianów W_1 i W_2 i udowodnijmy, że twierdzenie o znajdowaniu największego wspólnego współczynnika jest poprawne.

$W_1(x) = x^2(x-1)(x-2) = x^4 - 6x^3 + 13x^2 - 12x + 4$

$W_2(x) = (x-1)(x-2)(x-3) = x^3 - 6x^2 + 11x - 6$

Podzielmy teraz wielomian W_1 przez wielomian W_2.

$$
\begin{array}{l}
x \\
\hline
x^4 - 6x^3 + 13x^2 - 12x + 44 : (x^3 - 6x^2 + 11x - 6) \\
\underline{- x^6 + 4x^4 + 2x^3 - 3x^2 - 2x} \\
x^2 - 3x + 2x
\end{array}
$$

Widzimy, że otrzymana reszta wielomianu, nie jest wielomianem stopnia zerowego. Dokonujemy więc kolejnego dzielenia, przy czym nasz dotychczasowy dzielnik staje się nową dzielną, a otrzymana reszta dzielnikiem.

$$
\begin{array}{l}
x - 3 \\
\hline
x^3 - 6x^2 + 11x - 6 : (x^2 - 3x + 2) \\
\underline{- x^3 + 3x^2 - 2x} \\
- 3x^2 + 9x - 6 \\
\underline{3x^2 - 9x + 6} \\
0
\end{array}
$$

Otrzymaną resztą z dzielenia jest wielomian zerowy, zatem dzielnik tego działania jest największym wspólnym dzielnikiem dla wielomianów W_1 i W_2.

W tym miejscu warto wspomnieć o istnieniu NWW wielomianów, czyli ich najmniejszej wspólnej wielokrotności. Zapoznajmy się z jego definicją.

Definicja 10
Najmniejszą wspólną wielokrotnością dwóch wielomianów W_1 i W_2 nazywamy taki wielomian W_3, który jest wielokrotnością wielomianów W_1 i W_2 o możliwie najmniejszym stopniu.

Należy zauważyć, że istnieje tylko jeden taki wielomian, jeżeli założymy, że współczynnik stojący przy najwyższej potędze jest równy 1. Jest to termin ścisle związany z NWD. Potwierdza związane z nim twierdzenie.

Twierdzenie 10 *Aby obliczyć najmniejszą wspólną wielokrotność dwóch wielomianów W_1 i W_2, należy obliczyć wartość ilorazu z ich iloczynu i ich najmniejszej wspólnej wielokrotności.*

Dodatkowo należy podkreślić, że NWW dwóch wielomianów posiada wszystkie pierwiastki, które posiadał dowolny z nich. Krotność jej dowolnego pierwiastka jest równa jego maksymalnej krotności dla wielomianów, dla których została obliczona.

2.1.4 Dodatkowe twierdzenia dotyczące wielomianów

Istnieje mnóstwo twierdzień dotyczących wielomianów. Zapoznajmy się z tymi, które ułatwią nam znajdowanie pierwiastków wielomianów. Zapoznajmy się z pierwszym twierdzeniem, mówiącym o możliwości zamienienia dowolnego wielomianu o współczynnikach wymiernych, w wielomian o współczynnikach całkowitych.

Twierdzenie 11
Dowolny wielomian $W_1(x) = \frac{a_n}{b_n}x^n + \frac{a_{n-1}}{b_{n-1}}x^{n-1} + ... + \frac{a_1}{b_1}x + \frac{a_0}{b_0}$, o współczynnikach wymiernych, można przekształcić w wielomian $W_2(x) = k \cdot W_1(x)$, o współczynnikach całkowitych i tych samych pierwiastkach, co wielomian W. Wówczas:
$k = m \cdot NWW(b_0, b_1, ..., b_{n-1}, b_n)$, *gdzie* $m \in Z$

Twierdzenie to oznacza, że dysponując wyłącznie współczynnikami, będącymi liczbami całkowitymi, jesteśmy w stanie przedstawić dowolny wielomian o współczynnikach wymiernych. Przejdźmy teraz do twierdzenia, mówiącego o wartości wielomianu w danym punkcie.

Twierdzenie 12
Jeżeli wielomian W(x) podzelimy przez dwumian $x-x_0$, to reszta z tego dzielenia jest równa wartości tego wielomianu dla $x = x_0$.

W szczególnym przypadku reszta ta może być równa zero. Oznacza to, że liczba x_0 jest pierwiastkiem tego wielomianu. Wynika z tego bezpośrednio kolejne twierdzenie, znane jako twierdzenie Bezout.

Twierdzenie 13 (Bezout)
Liczba x_0 jest pierwiastkiem wielomianu $W(x)$ wtedy i tylko wtedy, gdy wielomian jest podzielny przez dwumian $x - x_0$.

Przejdźmy teraz do twierdzenia mówiącego o pierwiastkach wielokrotnych, bazującego na twierdzeniu Bezout.

Twierdzenie 14
Liczba x_0 jest pierwiastkiem k-krotnym wielomianu $W(x)$ wtedy i tylko wtedy, gdy wielomian jest podzielny przez $(x - x_0)^k$ i nie jest podzielny przez $(x - x_0)^{k+1}$.

Jedyną różnicą powyższego twierdzenia, w porównaniu do twierdzenia Bezout, jest to, że wielomian W przedstawiamy jako: $W(x) = (x-x_0)^k \cdot Q(x)$. Zapoznajmy się z twierdzeniem, mówiących o możliwej wartości pierwiastków, o ile są one całkowite.

Twierdzenie 15
Dany jest wielomian $W(x) = x^n + a_{n-1}x^{n-1} + ... + a_1x + a_0$, o współczynnikach całkowitych. Jeżeli wielomian W posiada pierwiastki całkowite, to są one dzielnikami wyrazu wolnego a_0.

Przejdźmy teraz do twierdzeń mówiących o możliwości rozłożenia wielomianu na czynniki.

Twierdzenie 16
Każdy wielomian $W(x)$ jest iloczynem czynników stopnia co najwyżej drugiego.

Oznacza to, że każdy wielomian stopnia co najmniej trzeciego jest rozkładalny na czynniki. Z powyższego twierdzenia wynika kolejne, mówiące, że rozkład ten jest zawsze jednoznaczny.

Twierdzenie 17
Niezerowy wielomian, o współczynnikach rzeczywistych, jest jednoznacznie rozkładalny na czynniki liniowe lub nierozkładalne czynniki kwadratowe, o współczynnikach rzeczywistych zakładając, że wartość współczynnika stojącego przy najwyższej potędze jest równa 1.

Oznacza to, że nie da się rozłożyć jednego wielomianu na czynniki, na dwa różne sposoby, tzn. tak, by istniał chociaż jeden czynnik (lub jego proporcjonalny odpowiednik), nie występujący w drugim rozkładzie. Ważnym aspektem, wynikającym z powyższego twierdzenia jest mowa, o tym że nie wszystkie wielomiany da się rozłożyć na czynniki liniowe, o współczynnikach całkowitych. Spójrzmy na pokazujący to przykład.

Przykład 8
Mamy dany wielomian $W(x) = x^3 - 1$. Rozłóżmy go na czynniki. Wiemy, że pierwiastkiem tego wielomianu jest $x_0 = 1$, zatem możemy przedstawić wielomian W jako iloczyn wielomianu $x - 1$ oraz drugiego wielomianu.
$W(x) = x^3 - 1 = x^3 - x^2 + x^2 - x + x - 1 = (x - 1)x^2 + (x - 1)x + x - 1 =$

$= (x-1)(x^2+x+1)$
Jest to równanie kwadratowe, więc najłatwiejszym sposobem na znalezienie jego pierwiastków jest obliczenie jego wyróżnika, zwanego też deltą. Jest on oznaczany jako Δ *i dla funkcji kwadratowej postaci* ax^2+bx+c *jest równy* b^2-4ac.
$\Delta = 1^2 - 4 \cdot 1 \cdot 1 = 1 - 4 = -3 < 0$ – *brak pierwiastków rzeczywistych*
Jak widać drugi z czynników jest właśnie przykładem nierozkładalnego czynnika kwadratowego, o współczynnikach rzeczywistych.

Powyższy czynnik da się rozłożyć na dwa czynniki liniowe, o współczynnikach zespolonych. W pracy tej będziemy jednak mówić wyłącznie o współczynnikach rzeczywistych, najczęściej zawężając jeszcze zbiór potencjalnych współczynników do liczb wymiernych. Przejdźmy do kolejnego twierdzenia, wynikającego bezpośrednio z dwóch poprzednich.

Twierdzenie 18
Każdy wielomian stopnia nieparzystego, ma przynajmniej jeden pierwiastek rzeczywisty.

Oznacza to, że każdy wielomian stopnia nieparzystego jesteśmy w stanie przedstawić jako iloczyn dwóch czynników, z których jeden jest czynnikiem liniowym, a drugi czynnikiem stopnia parzystego, który z kolei być może da się dalej rozłożyć, na wielomiany, o mniejszych stopniach.

2.1.5 Eliminacja pierwiastków wielokrotnych

Ważnym aspektem obliczanie zer wielomianów jest eliminacja pierwiastków wielokrotnych. Jest ona niezbędna, by móc skorzystać z metody ciągu Sturma. Zapoznajmy się z twierdzeniem dotyczącym krotności pierwiastków pochodnej wielomianu.

Twierdzenie 19
Jeżeli liczba jest pierwiastkiem k-krotnym wielomianu W*, to jest pierwiastkiem* $(k-1)$*-krotnym pochodnej tego wielomianu.*

By lepiej zrozumieć twierdzenie, spójrzmy na poniższy przykład.

Przykład 9
Mamy dany wielomian $W(x) = x^3 + 2x^2 + x$*. Obliczmy teraz kolejne pochodne wielomianu* W*.*

$$W^{(1)}(x) = 3x^2 + 2 \cdot 2x + 1 = 3x^2 + 4x + 1$$
$$W^{(2)}(x) = 2 \cdot 3x + 4 = 6x + 4$$
$$W^{(3)}(x) = 6$$

Obliczmy teraz pierwiastki wielomianu W *i jego kolejnych pochodnych.*
$W(x) = x^3 + 2x^2 + x = x(x^2+2x+1) = x(x+1)^2$
$x_1 = 0,\ k_1 = 1,\ x_2 = -1,\ k_2 = 2$
$W(1)(x) = 3x^2 + 4x + 1$

By przedstawić rozłożyć powyższy wielomian na czynniki, obliczmy wyróżnik równania kwadratowego.
$\Delta = 4^2 - 4 \cdot 3 \cdot 1 = 16 - 12 = 4$
$\sqrt{\Delta} = 2$
$x_1 = \frac{-4-2}{2 \cdot 3} = \frac{-6}{6} = -1,\ k_1 = 1,\ _2 = \frac{-4+2}{2 \cdot 3} = \frac{-2}{6} = -\frac{1}{3},\ k_2 = 1$
$W^{(2)}(x) = 6x + 4 = 6(x + \frac{2}{3})$
$x_1 = -\frac{2}{3},\ k_1 = 1$
$W^{(3)}(x) = 6$ – *brak pierwiastków*

Jak widać powyższy przykład potwierdza zastosowanie przedstawionego twierdzenia. Widzimy, że krotność wszystkich pierwiastków ulega zmniejszeniu o jeden, w kolejnej pochodnej. Dodatkowo możemy zauważyć, że pochodna może zawierać także pierwiastki, których nie miał dany wielomian. Ma to miejsce w przypadku, gdy wielomian, posiada przynajmniej dwa różne pierwiastki. Potwierdza to poniższe twierdzenie.

Twierdzenie 20 *Liczba nowych pierwiastków pochodnej wielomianu W' (takich których nie posiadał wielomian W) jest równa liczbie różnych pierwiastków wielomianu pomniejszonej o jeden.*

Spróbujmy teraz skorzystać z przedstawionego twierdzenia w przykładzie dokonującym eliminacji pierwiastków wielokrotnych.

Przykład 10
Dany jest wielomian W, określony wzorem: $W(x) = x^6 - 6x^4 - 4x^3 + 9x^2 + 12x + 4$. Dokonajmy eliminacji pierwiastków wielokrotnych dla wielomianu W.
Obliczamy pochodną wielomianu.
$W'(x) = 6 \cdot x^5 - 4 \cdot 6x^3 - 3 \cdot 4x^2 + 2 \cdot 9x + 12 =$
$= 6x^5 - 24x^3 - 12x^2 + 18x + 12 = 6(x^5 - 4x^3 - 2x^2 + 3x + 2)$
Obliczmy teraz resztę z dzielenia wielomianu W przez wielomian W'.

$$\begin{array}{l} x \\ \hline x^6 - 6x^4 - 4x^3 + 9x^2 + 12x + 4 : (x^5 - 4x^3 - 2x^2 + 3x + 2) \\ \underline{- x^6 + 4x^4 + 2x^3 - 3x^2 - 2x} \\ - 2x^4 - 2x^3 + 6x^2 + 10x + 4 \end{array}$$

Kluczowa jest wartość reszty z dzielenia. Gdyby otrzymana reszta z dzielenia była wielomianem zerowym, to pochodna wielomianu była równocześnie $NWD(W, W')$. W tym przypadku tak jednak nie jest, więc wykonujemy analogiczną operację z tą różnicą, że nową dzielną jest dotychczasowy dzielnik, a reszta z wielomianu jest nowym dzielnikiem. Tę operację wykonujemy tak długo, jak otrzymana reszta jest niezerowego stopnia. W przypadku gdy jest ona jednocześnie wielomianem zerowym, to podobnie jak wyżej, aktualny dzielnik, jest naszym $NWD(W, W')$. W przypadku, gdy otrzymana reszta jest niezerowym wielomianem stopnia zerowego, to największym wspólnym dzielnikiem wielomianów jest pewna niezerowa stała.

2.1.6 Twierdzenie Sturma

Przeanalizujmy na początek sposób konstruowania ciągu Sturma dla wielomianu W. Pierwszym wyrazem ciagu jest sam wielomian W. Z kolei drugim wyrazem jest pochodna wielomianu W. Kolejne wyrazy ciągu Sturma wyznaczamy obliczając resztę z dzielenia dwóch poprzednich wyrazów. Dzieje się to aż, do uzyskania pierwszej reszty wielomianu, będącej wielomianem stopnia zerowego. Za każdym razem dzieląc wielomian stopnia n, przez wielomian stopnia m, gdzie $m < n$, mamy gwarancję, że wielomian będacy resztą tego dzielenia będzie stopnia mniejszego niż m. Korzystając z tego faktu, wiemy, że liczba wyrazów ciągu Sturma dla wielomianu W jest niewiększa od $\deg(W) + 1$.

Definicja 11
Ciągiem Sturma dla wielomianu W nazywamy ciąg wielomianów: $X_0, X_1, X_2, ...$, takich że $X_0 = W$, $X_1 = W'$, a kolejne wyrazy definiuje się jako wielomiany przeciwne do reszty z dzielenia dwóch poprzednich wyrazów, przy czym ostatnim wyrazem ciągu Sturma jest pierwszy wielomian stopnia zerowego.

Wiemy już jak definiować ciąg Sturma. Przeanalizujmy teraz jaki wpływ ma uzyskany ciąg Sturma na obliczanie pierwiastków wielomianu. By to zrobić, zapoznajmy się z kolejną definicją, mówiącą o liczbie zmian znaków ciagu Sturma.

Definicja 12
Liczbą zmian znaków ciągu Sturma dla wielomianu $W(x)$ w punkcie x, obliczamy zliczając liczbę zmian pomiędzy kolejnymi wyrazami, pomijac te o wartości równej zero w punkcie x. Liczbę zmian znaku w punkcie $x = x_0$ definiujemy jako wartość funkcji $Z(x_0)$.

Wiemy już jak obliczać liczbę zmian ciagu Sturma. Sprawdźmy zatem, jak przekłada się ona na liczbę pierwiastków w danym przedziale.

Twierdzenie 21
Jeżeli wielomian W(x) nie ma pierwiastków wielokrotnych, to liczba pierwiastków rzeczywistych w przedziale $a < x \leqslant y$) jest równa $Z(a) - Z(b)$.

W ogólności twierdzenie Sturma można zastosować dla przedziału $(-\infty, +\infty)$, dopuszczając w ciągu Sturma wartości niewłaściwe $+\infty$ oraz $-\infty$. Wówczas $Z(-\infty)$ będzie oznaczać liczbę zmian znaków w ciągu $W(-\infty), W_1(-\infty), W_2(-\infty), ..., W_m(-\infty)$, zaś $Z(+\infty)$ liczbę zmian w ciągu $W(+\infty), W_1(+\infty), W_2(+\infty), ..., W_m(+\infty)$.

Twierdzenie 22
Liczba różnych pierwiastków rzeczywistych wielomianu $W(x)$ jest równa $Z(-\infty) - Z(+\infty)$.

Stosując twierdzenie Sturma dla coraz mniejszych przedziałów możliwe jest wyznaczenie pierwiastków wielomianu z dowolną dokładnością. Sposób ten określony jest mianem metody ciągu

Sturma. Twierdzenie Sturma jest bardzo mocnym środkiem, używanym do znajdowania pierwiastków w określonym przedziale. Zapoznajmy się z twierdzeniem, mówiącym o ograniczeniach dotyczących maksymalnej liczby zmian znaków dla ciągu Sturma.

Twierdzenie 23
Liczba zmian znaków ciągu Sturma dla wielomianu $W(x)$ jest mniejsza od liczby wyrazów tego ciągu i nie większa od stopnia wielomianu $W(x)$.

Można więc zauważyć, że warunkiem wystarczającym i jednocześnie koniecznym do tego by wielomian $W(x)$ stopnia n, posiadający wyłącznie pierwiastki jednokrotne, posiadał n pierwiastków rzeczywistych jest to, by wartość ciągu Sturma wynosiła n dla $\lim_{x\to-\infty}$ oraz 0 dla $\lim_{x\to+\infty}$.

Warto zauważyć, że dla liczb dostatecznie dużych, co do wartości bezwzględnej, z uwzględnieniem wartości dążących do wartości niewłaściwych $+\infty$ oraz $-\infty$ liczba zmian znaków zależy wyłącznie od współczynnika stojącego przy najwyżej potędze wielomianu. Znajduje to potwierdzenie w poniższym twierdzeniu.

Twierdzenie 24
Jeżeli współczynnik stojący przy najwyższej potędze wielomianu jest większy od 0, to wartość tego wielomianu jest większa od 0 w punkcie $\lim_{x\to+\infty}$ oraz mniejsza od 0 w punkcie $\lim_{x\to-\infty}$. Z kolei, gdy współczynnik stojący przy najwyższej potędze wielomianu jest mniejszy od 0, to wartość tego wielomianu jest mniejsza od 0 w punkcie x_0 dążacym do $+\infty$ oraz większa od 0 w punkcie x_0 dążacym do $-\infty$.

Na podstawie powyższego twierdzenia można zauważyć, że nie ma potrzeby wyliczania wartości wyrazów ciągu sturma dla wartości x_0 zbliżonych do $-\infty$ lub $+\infty$. Dzieje się tak dlatego, że przy odpowiednio dużej wartości bezwzględnej x_0, znak wielomianu $W(x)$ w punkcie x_0 zależy wyłącznie od współczynnika stojącego przy najwyższyej potędze. Fakt ten ma duży wpływ na optymalizacje obliczeń, gdyż dla niektórych wielomianów możemy ograniczyć się jedynie do sprawdzenia znaku przy najwyższej potędze, pomijąc dzięki temu wiele niepotrzebnych obliczeń

Jeżeli zależy nam na obliczeniu wartości wielomianu, dla odpowiednio dużych x, będącego ilorazem dwóch wielomianów, możemy skorzystać z poniższego twierdzenia.

Twierdzenie 25
Dany jest wielomian $W_1(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + ... + a_{n-1}x + a_n$, gdzie $a_0 \neq 0$, stopnia n oraz wielomian $W_2(x) = b_0x^{n-1} + b_1x^{n-2} + b_2x^{n-3} + ... + b_{n-2}x + b_{n-1}$, gdzie $b_0 \neq 0$, stopnia $n-1$. Wówczas wartość z ilorazu $W_1(x)$ przez wielomian $W_2(x)$ przy x dążącym do $+\infty$ lub x dążącym do $-\infty$ jest równa $\frac{a_0}{b_0}$.

Jak widać jesteśmy w stanie ustalić znak, a nawet dokładną wartość, największego współczynnika wielomianu, będącego ilorazem dwóch innych wielomianów, wykonując proste dzielenie dwóch liczb, będących odpowiednio najwyższymi współczynnikami wielomianów – dzielnej i dzielnika.

2.2 Biblioteka MPIR

2.2.1 Podstawowe informacje

Wstęp

Mpir jest przykładem biblioteki napisanej w języku C, pozwalającą operować na liczbach o dowolnej wielkości i dokładności. Jej celem było dostarczenie narzędzia dla arytmetyki liczb, która nie jest wspierana przez podstawowe typy, wbudowane w języku C.

Kluczowy jest fakt, że biblioteka ta została zoptymalizowana w taki sposób, by w zależności od potrzeb używać odmiennych algorytmów. Stało się tak dlatego, że algorytm działania na małych liczbach z niewielką precyzją jest prosty i znacznie różni się od wymyślnej metody, używanej w przypadku wielkich liczb, z setkami cyfr po przecinku.

Warto zauważyć, że optymalizacja kodu odbywa się także w zależności od rodzaju procesora, na którym są wykonywane obliczenia. Biblioteka przeznaczona dla procesorów Intel i7 dostarcza inny kod niż dla procesora Pentium 4 lub Athlon. Dzięki temu, z pewnością, działa w sposób bardziej wydajny, kosztem wygody programistów, którzy jej używają. Nie są oni w stanie, w łatwy sposób, zapewnić optymalnego działania programu, niezależnie od sprzętu na którym jest on uruchamiany, co może znacznie utrudnić proces powstawania nowego oprogramowania, bazującego na bibliotece mpir.

Licencja

Biblioteka mpir jest przykładem wolnego oprogramowania, charakteryzującego się kilkoma podstawowymi założeniami. Pierwszą z nich jest wolność pozwalająca użytkownikowi na uruchamianie programu, w dowolnym celu. Drugą jest jego analiza oraz dostosowywania go do swoich potrzeb. Kolejną cechą jest możliwość dowolnego rozpowszechnianie kopii programu. Ostatnią zaś jest udoskonalanie programu i publiczne rozprowadzanie własnych ulepszeń, z których każdy będzie mógł skorzystać.

Nazewnictwo i typy

Biblioteka mpir posiada odpowiedniki dla wszystkich podstawowych typów liczbowych, które są charakterystyczne dla języka C. Odpowiednikiem typów całkowitych jest typ mpz_t, zaś odpowiednikiem liczb zmiennoprzecinkowych, typu float, jest mpq_t. Dla liczb całkowitych możemy łatwo zdecydować, czy interesuje nas liczba ze znakiem, czy bez niego. Oba przypadki posiadają prawie identycznie nazwane funkcje. Jedyną różnicą jest to, że dla zmiennych ze znakiem występuje przyrostek si, od wyrazu angielskiego wyrazu signed, a dla zmiennych bez znaku przyrostek ui, od wyrazu unsigned. Typ mpq_t daje użytkownikowi możliwość zdefiniowania dowolnej liczby, dającej się przedstawić w postaci ułamka, czyli dowolnej liczby wymiernej. Ograniczenie to spowodowane jest faktem, iż liczba taka składa się z dwóch liczb typu mpz_t. Pierwsza z nich jest licznikiem, a druga mianownikiem.

Funkcje

W bibliotece przyjęto konwencję charakterystyczną dla języka C, polegającą na tym, że zarówno argumenty wejściowe jak i wyjściowe funkcji są przekazywane za pomocą referencji do funkcji, która zazwyczaj jest typu void, czyli nie zwraca żadnej wartości. Warto zauważyć, że argumenty wyjściowe są podawane przed wejściowymi. Stało się tak poprzez zachowanie analogii do operatora przypisania, w którym to zmienna wynikowa znajduje się po lewej stronie. W bibliotece mpir zmienna po zadeklarowaniu nie jest natychmiastowo gotowa do użycia, ponieważ wymaga inicjalizacji, przydzielającej jej miejsce w pamięci. Możemy to zrobić poprzez wywołanie odpowiedniej funkcji init, w argumencie podając odpowiednią zmienną. Analogicznie, aby zwolnić zajmowany przez zmienną, obszar pamięci należy wywołać funkcję clear. Istnieją także funkcje realokujące, czyli zwiększające lub zmniejszające, zajmowany obszar. Użycie ich jest jednak opcjonalne, gdyż zmiana rozmiaru jest automatycznie wykrywana. Raz zainicjalizowaną zmienną, można używać dowolną liczbę razy. Twórcy biblioteki zalecają unikanie nadmiernego korzystania z funkcji init i clear, ponieważ są to operacje czasochłonne i zbyt duża liczba ich wywołań może mieć negatywny wpływ na wydajność naszego programu.

Zarządzanie pamięcią

Biblioteka mpir, alokując w pamięci nowe obiekty, stara się minimalizować wielkość zajmowanego przez nie miejsca. Dodatkowa pamięć jest przyznawana dopiero, kiedy jej aktualna wielkość jest niewystarczająca. W przypadku zmiennych typów mpz_t i mpq_t, raz zwiększony rozmiar danej zmiennej nigdy nie zostanie zmniejszony. Zazwyczaj jest to najlepsza praktyka, ponieważ częsta alokacja pamięci ma niekorzystny wpływ na wydajność programu. Jeżeli aplikacja potrzebuje zmniejszyć zajmowane przez daną zmienną miejsce, można dokonać realokacji albo całkowicie zwolnić zajmowane przez nią miejsce. Dodatkowo kolejne zwiększania rozmiaru przeznaczonego na poszczególne zmienne może powodować zauważalny spadek wydajności lub fragmentację danych. Nie da się tego uniknąć, jeżeli chcemy zaalokować więcej miejsca dla danej zmiennej, w momencie, gdy w jej sąsiedztwie znajdują się już inne zmienne. Wówczas zależnie od implementacji mamy do czynienia z jednym z dwóch wariantów. W pierwszym przypadku, alokujemy nowe miejsce w pamięci, uzupełniając odpowiednio jego wartość, a następnie zwalniając wcześniej zajmowane miejsce. Alternatywą jest pozostawienie zajmowanej już pamięci i przydzielenie dodatkowej w innym miejscu. W takim przypadku jesteśmy zmuszeni do odpowiedniego zarządzania nieciągłymi fragmentami pamięci, co również może mieć bardzo negatywny wpływ na wydajność aplikacji. Z kolei, zmienne typu mpf_t mają niezmienny rozmiar, zależny od wybranej precyzji.
Warto zauważyć, że biblioteka standardowo do alokowania pamięci używa funkcji malloc, ale możliwe jest także korzystanie z funkcji alloca. Pierwszy alokator zajmuje się tylko przydzieleniem pamięci, a użytkownik musi sam pamiętać, o jego zwolnieniu, w odpowiednim momencie. Z kolei, drugi z nich zdejmuje tę odpowiedzialność ze strony programisty i sam dba o zwolnienie pamięci, gdy jest ona już nie używana.

Kompatybilność

Poza kilkoma wyjątkami, biblioteka mpir jest kompatybilna z odpowiednimi wersjami biblioteki gmp. Dodatkowo, jej twórcy starają się nie usuwać istniejących już funkcji. W przypadku, gdy któraś przestanie być rekomendowana, zostaje po prostu zaznaczona jako przestarzała, ale jej implementacja nie przestaje istnieć w kolejnych wydaniach biblioteki. Dzięki temu program, bazujący na starszej wersji biblioteki, zadziała też na nowszej edycji.

Wydajność

Dla małych liczb, narzut na korzystanie z biblioteki może być znaczący w porównaniu do typów prostych. Jest to nieuniknione, ale celem biblioteki jest próba znalezienia złotego środka pomiędzy wysoką wydajność, zarówno dla małych, jak i dużych liczb.

Operacje w miejscu

Operacje obliczania wartości bezwzględnej i negacji danej liczby są bardzo szybkie, gdy obliczane są w miejscu, tzn. wtedy, gdy zmienna wejściowa jest również zmienną wyjściową. Wówczas nie ma potrzeby alokacji i zwalniania pamięci, a cała funkcja sprowadza się do ustawienia odpowiedniego bitu, mówiącego o znaku liczby. Według specyfikacji biblioteki mpir, zauważalny powinien być także zysk, w przypadku operacji dodawania, odejmowania i mnożenie w miejscu. W momencie, gdy drugim argumentem jest nieduża liczba całkowita, operacje te są nieskomplikowane i bardzo szybkie.

2.2.2 Instalacja

Instalacja biblioteki mpir jest różna w zależności od systemu operacyjnego. W systemach uniksowych instalacja polega na zbudowaniu i instalacji, korzystając ze źródeł. Pod Windowsem jest ona równie łatwa i przebiega analogicznie, o ile używamy cygwina lub mingw. Są to narzędzia, które zapewniają, programom działającym pod systemem Windows, funkcjonalność przypominającą system Linux. Bardziej skomplikowane będzie użycie biblioteki bez wyżej wymienionych narzędzi, ale warto zaznaczyć, że biblioteka ta może być budowana z użyciem Microsoft Visual Studio, począwszy od wersji 2010, korzystając z programu o nazwie yasm asembler. Dodatkowo korzystanie z biblioteki różni się w zależności od tego czy potrzebujemy jej statyczną (mpir/lib) czy dynamiczną (mpir/dll) wersję.

Aby używać biblioteki mpir w naszym programie musimy do naszego programu dodać następującą linię:

```
#include <mpir.h>
```

Pozwoli ona nam na używanie wszystkich funkcji i typów, które udostępnia dla nas biblioteka. Dodatkowo wymagana jest kompilacja z dołączeniem naszej biblioteki, poprzez dodanie opcji -lmpir, np.:

```
gcc myprogram.c -lmpir
```

Jeżeli chcemy skorzystać z biblioteki, wspierającej język C++ dodatkowo musimy dodać także opcję -mpirxx, np.:

```
g++ myprogram.cc -lmpirxx -lmpir
```

2.2.3 Operacje na liczbach całkowitych

W niniejszym podrozdziale omówię podstawowe funkcje liczbowe, które mają zastosowanie zarówno dla liczb całkowitych typu mpz_t, jak i rzeczywistych typu mpq_t.

Funkcje inicjalizujące

```
void mpz_init (mpz_t integer)
```

Alokuje w pamięci miejsce na zmienną integer i ustawia jej wartość na 0.

```
void mpz_clear (mpz_t integer)
```

Zwalnia miejsce zajmowaną przez zmienną. Funkcja ta powinna być używana dla każdej zmiennej, w momencie, gdy nie ma już potrzeby, by z niej korzystać.

```
void mpz_realloc2 (mpz_t integer, mp bitcnt_t n)
```

Zmienia rozmiar zajmowanego przez zmienną miejsca. Funkcja jest używana z wartością n, większą od aktualnej, by zagwarantować zmiennej określone miejsce w pamięci. Z drugiej strony, zostaje wywołana z wartością mniejszą, jeżeli chcemy zmniejszyć liczbę zajmowanego przez zmienną miejsca. Gdy nowy rozmiar jest wystarczający by pomieścić aktualną wartość zmiennej, to zostaje ona zachowana. W przeciwnym razie zostanie ona ustawiona na 0. Istnieje również przestarzała funkcja mpz_realloc, ale jej użycie jest niezalecane. Nie została usunięta, by zachować kompatybilność wsteczną.

Funkcje przypisania

```
void mpz_set (mpz_t rop, mpz_t op)
```

Pozwala na przypisanie wartości tych samów typów. Wartość zmiennej op jest ustawiana jako rop. Jest to odpowiednik operatora przypisania.

```
void mpz_set_sx (mpz_t rop, intmax_t op)
```

Pozwala na przypisanie zmiennej typu całkowitego do wartości typu mpz_t.

```
void mpz_set_d (mpz_t rop, double op)
```

Zapisuje wartość typu double do zmiennej typu mpz_t.

```
void mpz_set_q (mpz_t rop, mpq_t op)
```

Pozwala na zrzutowanie liczby rzeczywistej typu mpq_t do zmiennej mpz_t. Gdy wartość mpq_t nie jest liczbą całkowitą, zostaje ona zaokrąglona w dół, poprzez obcięcie jej części ułamkowej.

```
int mpz_set_str (mpz_t rop, char *str, int base)
```

Konstruuje liczbę typu mpz_t, na podstawie podanego łańcucha znaków, reprezentującego daną wartość. Zmienna base mówi o podstawie podanej liczby. Dopuszcza się występowanie białych znaków, które są ignorowane. Z kolei zmienna base dopuszcza 0 oraz wartości z zakresu $(2; 61)$. Gdy jest ona równa 0, podstawowa zostaje ustalona, bazując na początkowych znakach. Dla prefixów $0x$ oraz $0X$, reprezentacja liczby zostaje ustalona jako szesnastkowa. Gdy jest ona równa $0b$ lub $0B$, to liczba ta jest binarna, a gdy rozpoczyna się od zera, a drugi znak jest inny od wymienionych, to zostaje uznana za liczbę oktalną. W pozostałych wypadkach jest to liczba dziesiętna. Kolejne wartości dziesiętne od 10 do 35 są reprezentowane jako litery od a do z, przy czym nie ma rozróżnienia ze względu na wielkość liter. W przypadku wyższych wartości podstawy, wielkość liter ma znaczenie, przy czym liczby od 10 do 35 reprezentowane są przez duże litery, a liczby od 36 do 61 przez małe. Funkcja dokonuje weryfikacji, czy podany ciąg znaków w całości reprezentuje poprawną liczbę. Jeżeli tak, to zwraca ona wartość 0. W przeciwnym wypadku jest to 1.

```
void mpz_swap (mpz_t rop1, mpz_t rop2)
```

Używana jest do zamiany wartości pomiędzy dwoma zmiennymi typu mpz_t. Jej użycie jest rekomendowane, ze względów wydajnościowych. W przypadku braku tej funkcji, potrzeba by zmiennej tymczasowej. W tym celu musiałaby ona na początku zostać zaalokowana w pamięci, a na końcu zwolniona. Obie operacje są czasochłonne i zaleca się minimalizować ich użycie, więc użycie zoptymalizowanej funkcji służącej do zamiany wartości dużych liczb wydaje się zdecydowanie najlepszą i najszybszą opcją.

Funkcje konwersji

```
intmax_t mpz_get_sx (mpz_t op)
```

Rzutuje zmienną typu mpz_t na int. Jeżeli wartość jest poza zakresem liczb typu int, to rzutowanie następuje poprzez pozostawienie najmniej znaczącej części. Powoduje to, że funkcja w wielu przypadkach może okazać się bezużyteczna.

```
double mpz_get_d (mpz_t op)
```

Pozwala zrzutować typ mpz_t na zmienną typu double. Jeżeli jest to konieczne, stosowane jest zaokrąglenie. W przypadku, gdy eksponent jest za duży, zwrócony wynik jest zależny od danego systemu. Jeżeli jest dostępna, zwrócona może być wartość nieskończoności.

```
char* mpz_get_str (char *str, int base, mpz_t op)
```

Zwraca ciąg znaków reprezentujących liczbę op, o podstawie danej w parametrze o nazwie base. Funkcja ta jest analogiczna do mpz_set_str. Jeżeli parametrem str jest NULL, to ciąg znaków zostaje zwrócony przez funkcję. W przeciwnym razie funkcja umieszcza go pod adresem, na który

wskazuje zmienna str. Należy zadbać o to, by bufor do zapisania rezultatu funkcji był wystarczająco wielki. Powinien on być 2 bajty dłuższy niż długość zwróconej liczby w danym systemie liczbowym. Pierwszy bajt jest przeznaczony na wpisanie ewentualnego znaku minus, natomiast drugi na znak '\0' kończący łańcuch. Długość zasadniczej części liczbowej można pobrać używając funkcji mpz_sizeinbase (op, base), w której podajemy daną liczbę oraz podstawę.

Funkcje arytmetyczne

```
void mpz_add (mpz_t rop, mpz_t op1, mpz_t op2)
```

Ustawia wartość *rop* jako sumę op1 i op_2.

```
void mpz_sub (mpz_t rop, mpz_t op1, mpz_t op2)
```

Ustawia wartość *rop* jako różnicę op1 i op_2.

```
void mpz_mul (mpz_t rop, mpz_t op1, mpz_t op2)
```

Ustawia wartość *rop* jako iloczyn op1 i op_2.

```
void mpz_addmul (mpz_t rop, mpz_t op1, mpz_t op2)
```

Oblicza wartość iloczynu czynników op_1 i op_2, a następnie zwiększa rop o otrzymany wynik. Tożsame z wykonaniem działania $rop = rop + op_1 * op_2$.

```
void mpz_submul (mpz_t rop, mpz_t op1, mpz_t op2)
```

Oblicza wartość iloczynu czynników op_1 i op_2, a następnie zmniejsza rop o otrzymany wynik. Tożsame z wykonaniem działania $rop = rop - op_1 * op_2$.

```
void mpz_neg (mpz_t rop, mpz_t op)
```

Jako rezultat ustawia liczbę przeciwną do danej. Gdy rop i op są tą samą zmienną to mamy do czynienia z przykładem operacji w miejscu. Wówczas całe funkcja jest bardzo szybka, gdyż opiera się tylko na zmianie bitu, mówiącego o znaku danej liczby.

```
void mpz_abs (mpz_t rop, mpz_t op)
```

Funkcja w swoim działaniu jest bardzo podobna do funkcji mpz_neg. Jedyną różnicą jest to, że bit znaku nie zostaje zanegowany, a ustawiony tak, by wskazywał na wartość nieujemną.

Funkcje dzielenia

Używając niżej wymienionych funkcji należy pamiętać, że dzielenie przez 0, zarówno w przypadku obliczania wartości ilorazu, jak i reszty z dzielenia jest nielegalne. W przypadku, gdy wystąpi taka sytuacja, biblioteka zachowa się jak w przypadku dzielenia przez 0, dla liczb typu int, tzn. skończy działanie funkcji komunikatem o błędzie.

```
void mpz_tdiv_q (mpz_t q, mpz_t n, mpz_t d)
```

Funkcja oblicza iloraz z liczb n i d. Litera przed 'div' w nazwie funkcji mówi, o jej zachowaniu w przypadku, gdy rezultat nie jest liczbą całkowitą. Symbol 't' ('truncate'), oznacza obcięcie części ułamkowej.

```
void mpz_tdiv_r (mpz_t r, mpz_t n, mpz_t d)
```

Oblicza wartość reszty z dzielenia liczb n i d. Podobnie jak w funkcji mpz_tdiv_q, mamy do czynienia z przypadkiem obcięcia ewentualnej części ułamkowej ilorazu. Oznacza to, że zwracana reszta zawsze będzie tego samego znaku co zmienna n. Rekomendowana, gdy wartość ilorazu z dzielenia nie jest istotna.

```
void mpz_tdiv_qr (mpz_t q, mpz_t r, mpz_t n, mpz_t d)
```

Łączy dwie powyższe funkcje. Ze względów wygody i wydajności, zaleca się jej użycie wówczas, gdy potrzebujemy obliczyć zarówno iloraz, jak i resztę z dzielenia. Ważne jest by pamiętać, że przekazane argumenty q i r, muszą być referencjami do różnych zmiennych. W przeciwnym razie, działanie funkcji będzie niepoprawne, a wynik najprawdopodobniej błędny.

Oprócz wyżej wymienionych funkcji, istnieją także ich odpowiedniki, o odmiennym zachowaniu w kwestii zaokrąglania części ułamkowej ilorazu. Pierwszym z nich jest grupa funkcji, posiadająca w nazwie 'cdiv', gdzie litera 'c' ('ceil') oznacza sufit. Powoduje to, że w razie potrzeby, iloraz zostanie zaokrąglony w górę. Wówczas otrzymana reszta r będzie przeciwnego znaku do liczby d. Funkcjami z tej grupy są: mpz_cdiv_q, mpz_cdiv_r oraz mpz_cdiv_qr. Drugim odpowiednik jest grupa funkcji, posiadająca w nazwie 'fdiv', gdzie 'f' ('floor') oznacza podłogę. W tym przypadku, w razie potrzeby wynik zostanie zaokrąglony w dół. Reszta r będzie w tym przypadku tego samegu znaku co liczba d. Przykładami funkcji z tej grupy są: mpz_fdiv_q, mpz_fdiv_r oraz mpz_fdiv_qr. Zauważmy, że we wszystkich trzech przypadkach zostaje spełnione równanie: $n = q*d+r,\ gdzie\ 0 \leqslant |r| < |d|$.

```
int mpz_divisible_p (mpz_t n, mpz_t d)
```

Funkcja sprawdza, czy d dzieli liczbę n. Jeżeli nie, to zwracane jest 0, a w przeciwnym wypadku wartość od niego różna.

```
void mpz_divexact (mpz_t q, mpz_t n, mpz_t d)
```

Oblicza iloraz z dzielenia liczb n przez d. Działa poprawnie tylko w przypadku, gdy n jest podzielna przez d. By to sprawdzić można użyć funkcji mpz_divisable_p. Jest znacznie szybsza niż pozostałe funkcje, umożliwiające obliczanie ilorazu z dzielenia. Dlatego zawsze, gdy wiemy, że spełniony jest powyższy warunek, użycie tej funkcji jest wysoko rekomendowane.

Funkcje porównania

```
int mpz_cmp (mpz_t op1, mpz_t op2)
```

Porównuje wartości dwóch liczb. Zwraca wartość dodatnią, gdy $op_1 > op_2$, ujemną, gdy $op_1 < op_2$, oraz 0, gdy obie wartości są równe.

```
int mpz_cmpabs (mpz_t op1, mpz_t op2)
```

Porównuje wartość bezwzględną dwóch liczb. Zwraca wartość dodatnią, gdy $|op_1| > |op_2|$, ujemną, gdy $|op_1| < |op_2|$, oraz 0, gdy obie wartości są równe lub przeciwne.

```
int mpz_sgn (mpz_t op)
```

Określa znak danej liczby. Zwraca 1 dla wartości dodatnich, -1 dla ujemnych oraz 0 w przypadku zera. Należy zwrócić uwagę, że w obecnej implementacji mpz_sgn jest makrem i rozpatruje dany argument wielokrotnie.

Pozostałe funkcje

```
void mpz_pow_ui (mpz_t rop, mpz_t base, mpir_ui exp)
```

Podnosi liczbę base do potęgi exp.

```
void mpz_sqrt (mpz_t rop, mpz_t op)
```

Oblicza pierwiastek całkowity dla danej liczby. W przypadku, gdy wynik nie jest liczbą całkowitą, część ułamkowa zostaje obcięta.

```
void mpz_sqrtrem (mpz_t rop1, mpz_t rop2, mpz_t op)
```

Jest bardzo podobna do funkcji mpz_sqrt. Poza obliczeniem wartości pierwiastka z danej liczby, zwraca także różnicę pomiędzy daną liczbą, a kwadratem tego pierwiastka. Wartości zwracane można określić równaniem:
$rop_1 = \sqrt{op}$
$rop_2 = op - (\sqrt{rop_1})^2$
Zauważmy, że jeżeli $rop_2 = 0$, to rop_1 jest pierwiastkiem kwadratotowym z op.

2.2.4 Operacje na liczbach wymiernych

W tym podrozdziale rozpatrzę funkcje dla zmiennych typu mpq_t. Skupię się w nim na funkcjach, które nie występują dla liczb całkowitych typu mpz_t, opisanych w poprzednim podrozdziale lub ich implementacja znacząco się różni, od już przedstawionej. Na początku warto zaznaczyć, że zmienne mpq_t reprezentują wyłącznie liczby wymierne. Jest to spowodowane ich implementacją. Każda liczba typu mpq_t składa się z dwóch liczb całkowitych typu mpz_t, reprezentujących licznik i mianownik.

```
void mpq_init (mpq_t dest_rational)
```

Funkcja alokuje miejsce w pamięci i inicjuje wartość danej liczby, ustawiając licznik na 0 i mianownik na 1.

```
void mpq_canonicalize (mpq_t op)
```

Znajduje wspólne dzielniki licznika i mianownika, skracając ułamek. Dodatkowo zapewnia, że mianownik jest liczbą dodatnią, w razie potrzeby, mnożąc licznik i mianownik przez liczbę -1.

```
mpz_t mpq_numref (mpq t op)
```

Jest to makro, które dla podanego ułamka zwraca wartość jego licznika. Posiada odpowiednik w postaci funkcji, której użycie nie jest jednak zalecana ze względów wydajonościowych.

```
mpz_t mpq_denref (mpq t op))
```

Jest makrem, które zwraca wartość mianownika dla podanego ułamka. Działa analogicznie jak mpq_numref.

```
void mpq_set_num (mpq t rational, mpz t numerator)
```

Ustawia licznik podanego ułamka na wskazaną wartość. Funkcja stanowi ekwiwalent dla wywołania metody mpz_set, wywoływanej dla licznika, uzyskanego przy pomocy makra. Jej działanie jest wolniejsze, dlatego jej użycie nie jest zalecane, ze zwględów wydajnościowych.

```
void mpq_set_den (mpq t rational, mpz t denominator)
```

Funkcja ustawia mianownik danego ułamka na podaną liczbę całkowitą. Jej działanie jest analogiczne jak w przypadku mpq_set_num.

```
int mpq_set_str (mpq_t rop, char *str, int base)
```

Tworzy liczbę na podstawie wartości podanej jako łańcuch znaków. Jej wartość podawana jest w postaci najpierw licznik, a następnie mianownik, na zasadach podobnym jak w funkcji mpz_set_str. Obie liczby oddziela znak operatora dzielenia '/'. Funkcja zwraca wartość 0, w przypadku, gdy cały łańcuch wejściowy jest poprawny oraz -1 w innym przypadku. Jeżeli argument base jest równy 0, to format liczbowy, dla licznika i mianownika, jest ustalany osobno. Oznacza to, że możemy podać liczby w dwóch różnych formatach, np. "$0xFF/256$". Warto zauważyć, że funkcja mpq_canonicalize nie jest wołana automatycznie, więc jeżeli podaliśmy ułamek, który chcemy skrócić, to musimy pamiętać o jej wywołaniu.

```
char * mpq_get_str (char *str, int base, mpq_t op)
```

Funkcja zwracająca liczbę w postaci ułamka, tzn. licznik i mianownik, oddzielony znakiem kreski ułamkowej, jest analogiczna do funkcji mpz_get_str. Bufor wyjściowy, w którym chcemy umieścić rezultat funkcji powinien być wystarczający – równy $mpz\_sizeinbase(mpq\_numref(op), base) + mpz\_sizeinbase(mpq\_denref(op), base) + 3$. Jeden dodatkowy bajt w porównaniu do funkcji mpz_get_str, spowodowany jest koniecznością umieszczenia w buforze kreski ułamkowej.

Rozdział 3

Opis rozwiązania

3.1 Podział na moduły

Aplikacja została podzielona na trzy główne moduły: statyczna biblioteka, aplikacja konsolowa oraz testy jednostkowe. Każdy z nich został zawarty w osobnym projekcie, stworzonym w programie Microsoft Visual Studio 2015. Moduły te zostały ze sobą odpowiednio powiązane, razem tworząc solucję. Pomiędzy modułami występują jasno określone zależności. Aplikacja konsolowa i projekt testowy są od siebie całkowicie niezależne, więc zmiany i błędy w jednym z nich, nie mają wpływu na drugi. Obie te części zależą od biblioteki statycznej, bezpośrednio się do niej odwołując. Aplikacja konsolowa udostępnia interfejs użytkownika do wspomnianej funkcjonalności, a framework testowy weryfikuje jej poprawność. Dodatkowo oba projekty są zależne od biblioteki mpir, która musi być dołączana w sposób dynamiczny, poprzez dodanie odpowiedniego pliku typu dll.

3.1.1 Statyczna biblioteka

Statyczna biblioteka zawiera wszystkie logiczne aspekty projektu, posiada odpowiednie klasy, zdefiniowane funkcje i zaimplementowane algorytmy, gotowe do wykorzystania i przetestowania. To w tej częsci projektu są zaprojektowane klasy pomocniczne – CharsConstants i StringManager oraz klasa Number, pozwalająca na łatwe korzystanie z dużych liczb wymiernych. Ta ostatnia jest warstwą pośrednią pomiędzy biblioteką mpir, a pozostałymi częściami projektu. Klasa ta, jako jedyna zależy w bezpośredni sposób od implementacji wspomnianej biblioteki. Innymi słowy, nawet w przypadku bardzo radykalnych zmian w bibliotece mpir, jest to jedyna klasa, wymagająca zmiany. Jest to bardzo wygodny i bezpieczny sposób na całkowite uniezależnienie pozostałej części od implementacji innego projektu, na którego kierunek rozwoju programista nie ma wpływu.

Moduł zawiera także najważniejszą część projektu, czyli klasy umożliwiające wykonanie działań na wielomianach. Istnieją dwie takie klasy – PolynomialMap i PolynomialVector, używajace odpowiednio mapy i tablicy, do przetrzymywania struktury wielomianów. Dla obu takich klas została zdefiniowana klasa abstrakcyjna Polynomial, która implementuje wspólne funkcje, a także definiuje interfejs dla klas pochodnych. Działanie obu klas jest analogiczne, ale zostało zaimplementowane

w odmienny sposób, co ma zasadniczny wpływ na wydajność rozwiązania, dla konkretnych typów wielomianów.

Część ta jest zależna od odpowiednio skompilowanej biblioteki mpir. W przeciwieństwie do aplikacji konsolowej i frameworku testowego, wystarczająca jest jej statycznie dołączana wersja. Warto zaznaczyć, że wersja ta jest inna dla różnych komputerów. Jest ona zależna od rodzaju procesora, ponieważ jej kod asemblerowy został zoptymalizowany, pod jego konkretny typ. Może to powodować różnice w wydajności biblioteki, ale nie powinno mieć żadnego wpływu na jej deterministyczne zachowanie. Dlatego jej działanie zawsze powinno być przewidywalne i takie samo, niezależne od rodzaju platformy, na której jest używana.

3.1.2 Aplikacja konsolowa

Aplikacja konsolowa pozwala, w łatwy dla użytkownika sposób, na tworzenie wielomianów i obliczanie wartości ich pierwiastków w zadanym przedziale. Użytkownik podaje odpowiednie dane wejściowe z linii poleceń, a te są następnie przetwarzane. Otrzymane wyniki są wypisywane na standardowe wyjście w przystępny sposób, tzn. w systemie dziesiętnym i z określoną precyzją.

Warto zaznaczyć, że użytkownik może wprowadzać wielomiany w dowolny sposób. Mogą być one zarówno w postaci sumy kolejnych współczynników, jak i skomplikowanych iloczynów, składających się z poszczególnych wielomianów. Jedynym ograniczeniem jest to, by wyrażenie było poprawne składniowo oraz zawierało wyłącznie całkowite współczyniki i naturalne potęgi. Nie powinno być to jednak przeszkodą w obliczaniu pierwiastków dla skomplikowanych wielomianów. Należy bowiem pamiętać, że każdy wielomian o współczynnikach wymiernych, a tylko takie będziemy rozważać w niniejszej pracy, da się przedstawić przy pomocy proporcjonalnych liczb całkowitych. Odpowiednie ułamki zwykłe, zawsze możemy w ten sposób zamienić, wyciągając z danego wielomianu czynnik, będący odwrotnością najmniejszej wspólnej wielokrotności, wszystkich współczynników. Jako, że czynnik ten będzie stałą, to przy obliczaniu pierwiastków, można go po prostu pominąć, gdyż nie ma on na nie żadnego wpływu.

Użytkownik może korzystać z aplikacji tak długo, jak będzie sobie tego życzył. Zapytania o kolejne wielomiany przeprowadzone są w pętli i trwają, dopóki użytkownik nie wprowadzi ciągu znaków, odpowiedzialnego za wyjście z programu. Szczegółowe informacje, o tym jak korzystać z aplikacji są podane w jego instrukcji.

3.1.3 Framework testowy

Testy jednostkowe umożliwiają użytkownikowi sprawdzenie, czy określona funkcjonalność działa bezbłędnie. Zgodnie z ich założeniami, są w nich zdefiniowe podstawowe testy, weryfikujące różne przypadki testowe. W tym przypadku jest nieco inaczej, ponieważ w postaci testów jednostkowych, zostały zawarte również bardziej skomplikowane testy funkcjonalne. Wśród nich znajdują się też takie, które testują główne zadanie projektu, czyli dla wielomianu, tworzonego na podstawie otrzymanego wejścia, w postaci łańcucha znaków, potrafią, z określoną precyzją, znaleźć jego wszystkie pierwiastki rzeczywiste.

Jako środowisko testowe wybrany został framework wbudowany w narzędzie Microsoft Visual Studio 2015. Rozważana była też biblioteka o nazwie Google Test. Jej istotną zaletą, była przenośność, ale ostatecznie koncepcja ta została odrzucona. Spowodowane było to faktem, iż cały projekt pisany był w Visual Studio, a łatwość i wygoda, uruchamiania w nim testów jednostkowych jest bardzo duża. Dzięki temu wyborowi każda, nawet najmniejsza zmiana, mogła być bardzo szybka przetestowana, bez potrzeby wprowadzania jakichkolwiek dodatkowych zmian, co znacznie przyspieszało proces implementacji i weryfikacji przyjętego rozwiązania.

3.2 Główne klasy

3.2.1 CharsConstants

Klasa CharsConstants została zaimplementowana w celu zapewnienia większej czytelności kodu. Ważnym aspektem było zdefiniowane i łatwe rozpoznawanie wszystkich, mogących występować w łańcuchu wejściowym, legalnych znaków. Takie podejście ogranicza możliwość popełnienia prostych, a trudnych do wykrycia błędów w przetwarzaniu otrzymanego tekstu, np. literówki. W przypadku, gdy są zdefiniowane odpowiednio stałe i funkcje, ewentualna literówka, skończy się zasygnalizowaniem błędu już na etapie kompilacji. Przypadek taki jest sygnalizowany i łatwy do naprawienia, dzięki czemu programista nie musi zastanawiać się dlaczego, poprawny, wydałoby się, kod nie działa. Poniżej przedstawiam strukturę klasy, wraz ze zdefiniowanymi w niej stałymi i statycznymi metodami.

```
class CharsConstants
{
static const char Space = ' ';
static const char Tab = '\t';
static const char NewLine = '\n';
static const char LeastDigit = '0';
static const char GreatestDigit = '9';
static const char LeastUppercase = 'A';
static const char GreatestUppercase = 'Z';
static const char LeastLowercase = 'a';
static const char GreatestLowercase = 'z';

public:
static const char Plus = '+';
static const char Minus = '-';
static const char Mul = '*';
static const char Div = '/';
static const char Exp = '^';
static const char OpeningParenthesis = '(';
static const char ClosingParenthesis = ')';
```

```
static const char Var = 'x';

static int CharToInt(char c);
static bool IsDigit(char c);
static bool IsLetter(char c);
static bool IsUppercase(char c);
static bool IsLowercase(char c);
static bool IsWhitespace(char c);
static bool IsPlus(char c);
static bool IsMinus(char c);
static bool IsMul(char c);
static bool IsDiv(char c);
static bool IsExp(char c);
static bool IsOpeningParenthesis(char c);
static bool IsClosingParenthesis(char c);
static bool IsVar(char c);
static bool IsOperator(char c);
static bool IsLegalValue(char c);
static bool IsLegalOpeningOperator(char c);
};
```

3.2.2 StringManager

Klasa StringManager oferuje przydatne funkcje, na łańcuchach znaków. Wykorzystywana jest w metodach klasy Parser, w celu łatwiejszej jego implementacji oraz zwiększenia czytelności. Poniżej deklaracja klasy StringManager i udostępnianych przez nią metod.

```
class StringManager
{
public:
        static string EmptyString();
        static bool IsEmptyString(string s);
        static char ReturnLastChar(string s);
        static bool LastCharIsADigit(string s);
        static bool LastCharIsALetter(string s);
        static bool LastCharIsADigitOrALetter(string s);
        static bool LastCharIsAdigitOrALetterOrAParenthesis(string s);
        static int FindFirst(string s, char c);
        static int FindLast(string s, char c);
        static string Substr(string s, int first, int last);
        static vector<string> Split(string s, string operators);
        static int FindClosingParenthesis(string s);
```

```
        static string ParenthesisContent(string s);
};
```

3.2.3 Parser

Parser to klasa umożliwiająca użytkownikom podanie wielomianów poprzez standardowe wejście. Składa się tylko z dwóch metod. Zadaniem pierwszej z nich jest unifikacja wielomianu podanego na wejściu. Drugą metodą klasy jest tworzenie obiektu, reprezentującego wielomian. Metoda UniformInputString jest wykonywana zawsze na początku funkcji ConvertToPolynomial. Następnie na podstawie otrzymanego wyniku, funkcja ta dokonuje kolejnych obliczeń. Argument type mówi o tym jakiego typu wielomian ma zostać stworzony wewnątrz funkcji. Zwracana referencja wskazuje na obiekt wybranego typu. Dla wartości 0 lub 1, wybierany jest odpowiedni typ – PolynomialMap lub PolynomialVector. Poniżej, krótka definicja klasy Parser oraz jej metod.

```
class Parser
{
public:
        string UniformInputString(string s);
        Polynomial& ConvertToPolynomial(string inputS, int type = 0);
};
```

3.2.4 Number

Klasa Number stanowi warstwę pośrednią, pomiędzy klasą wielomianu, a wykorzystaniem typów liczbowych, z biblioteki mpir. Została ona stworzona, by uniezależnić implementacje wielomianów od wykorzystanego sposobu reprezentacji dużych liczb. Dzięki temu jakakolwiek zmiana w udostępnianych przez bibliotekę mpir, klasach i funkcjach, wymaga zmiany kodu aplikacji, tylko w jednym miejscu. Ma to niebagatelny wpływ na łatwość utrzymania aplikacji. Dodatkowo, wprowadzając metody opakowujące funkcje biblioteczne, możliwe jest automatyczne wykonanie dodatkowych operacji i stworzenie nowych funkcji, ułatwiających tworzenie klas i zwiększających czytelność kodu. Narzut czasowy związany z koniecznością wywoływania funkcji pośrednich został oceniony jako niewielki, a ich wpływ na przeprowadzone testy i otrzymane rezultaty jako pomijalny. Zapoznajmy się teraz z definicją klasy Number.

```
class Number
{
public:
        mpq_t value;
        explicit Number();
        explicit Number(double value);
        Number(const Number &bigNumber);
        ~Number();
```

```
Number Neg();
Number Abs();
Number Copy();
void SetMaxNegativeValue();
void SetMaxValue();
bool IsPlusInfinity();
bool IsMinusInfinity();
bool IsInfinity();
bool IsVerySmallValue();
bool IsZero();
bool IsWithRequiredPrecision();
int IsInVector(vector<Number> v);

bool operator == (Number bigNumber);
bool operator != (Number bigNumber);
bool operator > (Number bigNumber);
bool operator < (Number bigNumber);
bool operator >= (Number bigNumber);
bool operator <= (Number bigNumber);
Number operator = (Number bigNumber);
Number operator + (Number bigNumber);
Number operator - (Number bigNumber);
Number operator * (Number bigNumber);
Number operator / (Number bigNumber);
Number operator ^ (int power);
Number operator += (Number bigNumber);
Number operator -= (Number bigNumber);
Number operator *= (Number bigNumber);
Number operator /= (Number bigNumber);
Number operator ^= (int power);
bool operator == (double value);
bool operator != (double value);
bool operator > (double value);
bool operator < (double value);
bool operator >= (double value);
bool operator <= (double value);
Number operator = (double value);
Number operator + (double value);
Number operator - (double value);
Number operator * (double value);
Number operator / (double value);
```

```
        Number operator += (double value);
        Number operator -= (double value);
        Number operator *= (double value);
        Number operator /= (double value);
        string ToString();
        string MakeNice(string result);
        string RoundNine(string result);
        string TruncateZero(string result);
        void Print();
}
```

Jak widać, przeciążone zostały wszystkie przydatne dla liczb operatory, zarówno dla klasy Number jak i dla typu double. W języku C++ wszystkie wbudowane typy liczbowe – int, long oraz float, można łatwo zrzutować na zmienną typu double. Wynika z tego, że przy pomocy powyższych operatorów, jesteśmy w stanie w łatwy sposób, dokonać dowolnego działania na dużych liczbach, niezależnie od typu drugiego operandu. Wyjątkiem jest operator potęgowania, dla którego możliwe jest podniesienie do dowolnej potęgi, pod warunkiem, że jej wartość jest liczbą naturalną.

Klasa posiada tylko jeden element – obiekt typu mpq_t, reprezentujący właściwą liczbę wymierną. W klasie zostały zdefiniowane podstawowe funkcje, takie jak obliczenie wartości bezwzględnej oraz liczby przeciwnej. W celu zwiększenia czytelności i łatwiejszej implementacji, zostały przeciążone wszystkie przydatne operatory. Rozpatrzmy dwa przykłady użycia biblioteki mpir – pierwszy z bezpośrednim użyciem dostępnych funkcji i drugi z wykorzystaniem klasy Number.

```
mpq_t simple_function(int a, int b)
{
        mpq_t value1, value2, value3, value4;
        mpq_inits(value1, value2, value3, value4);
        mpq_set_d(value1, (double) a);
        mpq_set_d(value2, (double) b);

        mpq_mul(value3, value1, value2);
        if (mpq_cmp(value1, value2) > 0)
        mpq_add(value4, value1, value3);
        else
        mpq_add(value4, value2, value3);

        mpq_clears(value1, value2, value3);
        return value4;
}

mpq_t simple_function_with_Number(int a, int b)
{
        Number& number1 = new Number(a);
```

```
        Number& number2 = new Number(b);
        Number& number3 = new Number(a*b)
        Number& number4 = new Number();

        if (number1 > number2)
        number4 = number1 + number3;
        else
        number4 = number2 + number3;

        delete(number1);
        delete(number2);
        delete(number3);
        return number4;
}
```

Łatwo zauważyć, że w drugim przypadku kod jest zdecydowanie czytelniejszy. Poprzez przeciążenie operatorów, operacje na dużych liczbach wymiernych wyglądają identycznie jak działania na wbudowanych typach liczbowych. Dzięki temu, wyeliminowano bezpośrednie wywoływanie funkcji bibliotecznych. Ich użycie, pomimo charakterystycznych nazw zawsze wymagało chwili zastanowienia nad kolejno przekazywanymi argumentami. W przypadku nieskomplikowanych funkcji, stosunkowo duży narzut związany jest z koniecznością inicjalizowania referencji i zwalniania miejsca, przez obiekty na które wskazują. Nie stanowi to jednak sporej zmiany, w stosunku do przykładu pierwszego, w którym, na początku funkcji nastąpiły deklaracja i inicjalizacja zmiennych, a na końcu ich zwolnienie z pamięci.

Należy zaznaczyć, że dzięki klasie Number, nie tylko uzależniamy się od zmian wewnątrz biblioteki mpir, ale możemy także z niej całkowicie zrezygnować. Możemy wówczas skorzystać z alternatywnej biblioteki lub całkowicie się uniezależnić, bazując w pełni na własnej implementacji. Ta ostatnia opcja była rozważana przeze mnie w początkowej fazie projektu. W wersji prototypowej powyższa koncepcja pozwoliła mi na reprezentację współczynników wielomianu, przy pomocy typu double. Jej zakres i precyzja była wystarczająca, by aplikacja działała poprawnie dla wielomianów niskich stopni, z ograniczoną precyzją wyszukiwania pierwiastków. Planowana była własna implementacja typu liczbowego, reprezentującego wysokie wartości dowolnej precyzji. Ostatecznie jednak koncepcja ta została odrzucona ze względów wydajnościowych. Takie rozwiązanie, napisane w języku C++, nie mogłoby się bowiem równać z optymalizowanym kodem biblioteki mpir, pisanym w języku C i posiadającym krytyczne funkcje w postaci asemblerowych wstawek.

3.2.5 Polynomial

Klasa Polynomial to klasa abstrakcyjna, posiadająca zdefiniowane wszystkie funkcje, niezbędne do znalezienia pierwiastków danego wielomianu. Jest ona klasą bazową dla klas PolynomialMap i PolynomialVector, reprezentujących wielomiany, przy pomocy odpowiedniej struktury. Pierwsza z nich bazuje na mapie, czyli strukturze opartej na parach (klucz, wartość). Klucze muszą być

różnowartościowe, co umożliwia jednoznaczne znalezienie wartości dla dowolnego z nich. Pozwala to na posiadanie informacji wyłącznie o niezerowych współczynnikach wielomianu. Druga z nich bazuje na wektorze, jako przykładzie tablicy, której kolejne elementy są położone w pamięci operacyjnej obok siebie. Pierwszy element tablicy to współczynnik wielomianu, stojący przy potędze zerowej, a następne wartości to współczynniki, stojące przy kolejnych, coraz to wyższych potęgach. Dzięki temu dostęp do dowolnego wyrazu wielomianu jest bardzo szybki. Jednocześnie jednak, konieczne jest przetrzymywanie informacji o wszystkich współczynnikach wielomianu, stojących przy kolejnych potęgach, od potęgi zerowej, aż do najwyższej potęgi z niezerowym współczynnikiem, równej stopniowi wielomianu.

Klasa Polynomial posiada zarówno już napisane metody, jak i takie, które zostały tylko zaprojektowane i oznaczone jako czysto wirtualne, czyli konieczne do zaimplementowania w klasie dziedziczącej. Do tego pierwszego zbioru zalicza się część metod, która nie odnosi się do konkretnej struktury wielomianu. Dzięki temu, w klasach podrzędnych nie ma konieczności ponownego ich pisania. Posiadanie jednej implementacji zamiast dwóch pozwalało na łatwiejszą i dużo sprawniejszą konstrukcję obu klas dziedziczących. Spore zmiany projektowe i niewielkie korekty wystarczyło wprowadzić tylko raz, zamiast niepotrzebnie je powielać. Niestety, w przypadku sporej części funkcji, mimo bardzo zbliżonej implementacji, nie było możliwe ich połączenie i umieszczenie w klasie bazowej. Spowodowane to było koniecznością odwołań do konkretnego typu danych, w którym zostały umieszczone wyrazy wielomianu. Uogólnione są natomiast wszystkie pozostałe funkcje, w tym także te, które posiadają wywołania funkcji czysto wirtualnych. W przypadku tym, zastosowany został polimorfizm, czyli jeden z paradygmatów programowania obiektowego. Pozwala on odwoływać się do zdefiniowanej metody, bez znajomości jej implementacji. Wywołana funkcja zadziała różnie, zależnie od typu obiektu, na którym zostanie wykonana. Z punktu widzenia poprawnego działania klasy Polynomial, bez znaczenia jest implementacja operatora przypisania, o ile spełnia on swoją rolę. W ten sposób możliwe jest działanie na obiekcie wielomianu, niezależnie, czy bazuje on na tablicy, czy mapie.

W języku C++, by można zastosować polimorfizm, poszczególne metody muszą operować na wskaźnikach lub referencjach. Nie możliwe jest użycie tego mechanizmu w przypadku przekazywaniu obiektów poprzez wartość. Wytłumaczenie tego jest bardzo proste i opiera się na zmiennym rozmiarze obiektów klas pochodnych. Uniemożliwia to odwołanie się do danego obiektu, póki nie znamy jego dokładnego typu. Inaczej jest w przypadku referencji i wskaźników, ponieważ ich rozmiar jest stały, a zmieniać się może jedynie ich wartość, czyli miejsce w pamięci, do którego się odwołują.

Z uwagi na powyższy fakt, w programie wszystkie obiekty klasy Polynomial są przekazywane poprzez referencję. By referencję, można przekazywać pomiędzy funkcjami, obiekt na który wskazuje, musi zostać stworzony dynamicznie. Można to zrobić poprzez użycie, znanej z języka C, funkcji malloc, alokującej miejsce w pamięci lub typowego dla języka C++, operatora new. W przeciwnym wypadku, zmienna będzie widoczna tylko w miejscu, w którym zostanie stworzona, np. wewnątrz funkcji. Próba odwołania się do takiej wartości, w miejscu, w którym zmienna nie jest widoczna, skończy się błędem czasu wykonania i komunikatem o naruszeniu dostępu. Stanie się tak, ponieważ referencja w takim przypadku będzie dalej istnieć, ale obiekt, na który wska-

zuje, już nie. Takie błędy często są popełniane przez niedoświadczonych programistów, a próba ich lokalizacji i znalezienia przyczyny, na pierwszy rzut oka, nie jest oczywista. W momencie, gdy skorzystamy z dynamicznego tworzenia obiektu, będzie on istniał tak długo, dopóki nie zostanie jawnie usunięty w kodzie programu lub aplikacja nie skończy swojego działania, zwalniając przy tym całą zajmowaną pamięć. Kiedy nie wszystkie stworzone obiekty zostają usunięte, dochodzi do tzw. wycieków pamięci. W zależności od ich rozmiarów i czasu działania aplikacji, ich konsekwencje mogą być bardzo różne. W skrajnym wypadku, może dojść do wyczerpania całej dostępnej pamięci. Gdy zaczyna jej brakować, system operacyjny zapisuje jej ostatnio nieużywaną część na dysku, by w razie potrzeby móc ją odczytać. Z uwagi na to, że pamięć operacyjna jest wielokrotnie szybsza od dysków twardych, taka operacja powoduje gigantyczne opóźnienia w pracy komputera. Gdy szybkość kolejnych alokacji pamięci jest większa, niż jej zrzucanie na dysk, w pewnym momencie komputer ulegnie całkowitemu zawieszeniu. Wówczas jedynym wyjściem jest, często bardzo niepożądany, restart systemu. Bezpiecznym rozwiązaniem jest ustalenie limitu na wykorzystanie przez pojedynczą aplikację. Może to znacznie ułatwić debugowanie i znalezienie ewentualnego błędu, a także zabezpieczy użytkownika przed uruchomieniem złośliwego oprogramowania, mającego na celu doprowadzenie do wspomnianej wyżej sytuacji.

Poniżej przedstawiam definicję klasy abstrakcyjnej Polynomial. Wyróżnić w niej można kolejne sekcje – zmienne klasy, konstruktory, funkcje i operatory. Te ostatnie możemy podzielić pomiędzy te, które posiadają implementacje w klasie abstrakcyjnej Polynomial oraz te, które zostały zdefiniowane jako czysto wirtualne i konieczne jest ich nadpisanie w klasie pochodnej.

```
class Polynomial
{
public:
        MAP m;
        VECTOR v;
        bool isNew = true;
        int type = 0;
        string inputS = "";
        vector<Number> roots;
        int id = 0;

        explicit Polynomial();
        explicit Polynomial(Number number);
        ~Polynomial();

        virtual Polynomial& CreatePolynomial() = 0;
        virtual Polynomial& CreatePolynomial(Number number) = 0;
        virtual void Clear() = 0;
        virtual bool IsZero() = 0;
        virtual int Size() = 0;
        virtual int PolynomialDegree() = 0;
```

```
virtual Number Value(int power) = 0;
virtual pair<Polynomial&, Polynomial&> \
DividePolynomials(Polynomial& p1, Polynomial& p2) = 0;
virtual void SetNumberValue(int power, Number number) = 0;
virtual int NumberOfChangesSign(Number a) = 0;
virtual Polynomial& NegativePolynomial() = 0;
virtual Polynomial& Derivative() = 0;
virtual Number PolynomialValue(Number a) = 0;
virtual string ToString() = 0;
virtual void SturmClear() = 0;
virtual int TheLowestNonZeroValue() = 0;

virtual bool operator==(Polynomial& p2) = 0;
virtual Polynomial& operator = (Polynomial& p2) = 0;
virtual Polynomial& operator + (Polynomial& p2) = 0;
virtual Polynomial& operator - (Polynomial& p2) = 0;
virtual Polynomial& operator * (Polynomial& p2) = 0;

virtual VECTOR VectorValuesExceptValueOfPolynomialDegree
    (int degree) { return{}; };
virtual MAP MapValuesExceptValueOfPolynomialDegree
    (int degree) { return{}; };

bool Set(string s);
bool IsNew();
PAIR ValueOfPolynomialDegree();

bool ValueEquals(int power, Polynomial& p2);
void SetValue(int power, double value);
void Add(int power, Number number);
void Sub(int power, Number number);
PAIR Mul(int power1, Number number1,
      int power2, Number number2);
PAIR Div(int power1, Number number1,
      int power2, Number number2);
Polynomial& Nwd(Polynomial& p1, Polynomial& p2);
Polynomial& PolynomialAfterEliminationOfMultipleRoots();
void Normalize();
Number CoefficientValue(PAIR pair1, Number a);
Number NextNumberFromRange(Number a, Number b);
int NumberOfRoots(Number a, Number b);
```

```
        int AddNextRoot(Number x);
        vector<Number> FindRoots(Number a, Number b);
        void PrintRoots(Number a, Number b);

        bool operator!=(Polynomial& p2);
        Polynomial& operator / (Polynomial& p2);
        Polynomial& operator % (Polynomial& p2);
        Polynomial& operator ^ (int power);
        Polynomial& operator += (Polynomial& p2);
        Polynomial& operator -= (Polynomial& p2);
        Polynomial& operator *= (Polynomial& p2);
        Polynomial& operator /= (Polynomial& p2);
        Polynomial& operator %= (Polynomial& p2);
        Polynomial& operator ^= (int power);
        void Print();
        void PrintInput();
};
```

3.2.6 PolynomialVector

Klasy PolynomialMap i PolynomialVector są implementacjami klasy abstrakcyjnej Polynomial. Obie one nadpisują wszystkie czysto wirtualne metody klasy bazowej. Wiele z tych metod wygląda bardzo podobnie, natomiast istotną różnicą jest konieczność odwołania się do konkretnej reprezentacji danych. To właśnie dlatego zostały one oznaczone w klasie bazowej, jako metody abstrakcyjne. Takie zaprojektowanie klasy, wymusza na użytkowniku korzystanie z interfejsu, udostępnionego przez klase bazowe. Jego poprawne zaimplementowanie daje gwarancje, że klasa będzie wykonywać to czego oczekuje bazowa, nie narzucajac sposóbu, w jaki ma to robić.

Jak zostało wspomniane, różnicą obu klas jest podejście do współczynników zerowych. Koncepcja reprezentacji wszystkich z nich została oparta na typie vector, z biblioteki STL. Tablica zakłada, że kolejne współczynniki bedą zapisywane w jej kolejnych komórkach, przy czym indeks w tablicy, począwszy od zerowego, określa potęgę dla współczynnika, którego wartość jest tam zapisana. Aplikacja musi pamiętać wskaźniki na pierwszy i ostatni element. Na ich podstawie jest ona w stanie stwierdzić, ile elementów zawiera, co w sposób bezpośredni przekłada się na stopień danego wielomianu. Największym mankamentem takiego podejścia jest konieczność posiadania informacji, o wszystkich współczynnikach, także zerowych. W przypadku wielomianów rzadkich, o wysokim stopniu, np. $x^{1000} - 1$, konieczne jest przechowywanie 999 zer w kolejnych komórkach. Poza negatywnym wpływem na złożoność pamięciową takiego podejścia, istotniejszym wydaje się fakt, że znalezienie wyłącznie niezerowych elementów, wymaga przejrzenia wszystkich komórek tablicy. Zatem, by dodać do siebie dwa wielomiany x^{1000} oraz $2x^{1000}$, konieczne jest wykonanie aż 1001 sumowań lub sprawdzenie dla każdego z nich, czy jest potrzebne.

Dodatkowo, w przypadku, gdy w wyniku działania, zmienia się stopień wielomianu, konieczne

jest rozszerzenie lub pomniejszenie tablicy, poprzez aktualizacje odpowiednich wskaźników. Jeżeli chodzi o złożoność czasową funkcji, nie ma to jednak wpływu, ponieważ dla wszystkich typów operacji, i tak jesteśmy zmuszeni przejrzeć wszystkie elementy. Wyjątkiem jest stwierdzenie, czy mamy do czynienia z wielomianem zerowym, którego złożoność jest stała, gdyż wartość ustalana jest na podstawie rozmiaru tablicy.

Jeżeli chodzi o zalety korzystania z tablicy, jako struktury do przechowywania współczynników wielomianu, to najważniejszy jest stały czas dostępu do dowolnego z nich, poprzez możliwość odwołania się do konkretnego elementu tablicy. W przypadku zapisu pod wskazany adres, czas ten może się wydłużyć, w przypadku zmiany stopnia wielomianu. Widać więc, że użycie tablicy wydaje się uzasadnione w przypadku operacji na wielomianach gęstych, których stopień wielomianu nie zmienia się zbyt często. Struktura ta doskonale sprawdzi się do zsumowania dwóch wielomianów, stopnia setnego, zawierających wszystkie współczynniki równe 1. Z kolei, dla przypadku, gdy wielokrotnie sumujemy rzadkie wielomiany, o przeciwnych współczynnikach, wydajność tej struktury prawdopodobnie nie będzie zadowalająca.

3.2.7 PolynomialMap

Klasa PolynomialMap zakłada strukturę wielomianu, reprezentującą wyłącznie niezerowe współczynniki. Pozwala to zaoszczędzić miejsce w pamięci, dla przechowywania wartości zerowych i łatwo stwierdzić ile ich jest. Również w tej klasie wykorzystałem typ z bibiolteki STL, jakim jest map. Mój wybór był spowodowany faktem, że jest to rodzaj kontenera danych, w którym mamy łatwy dostęp do żądanej wartości. Nie jest on tak szybki, jak wektor, bo w każdym zapytaniu, musimy dowiedzieć się o położeniu konkretnej wartości. Mapa nie gwarantuje nam, że trzymane w niej dane będą znajdować się, w położonej blisko siebie pamięci. Dzięki temu, modyfikacja jej rozmiaru, jest bardzo łatwa, gdyż wystarczy zmienić tylko te wartości, które rzeczywiście się zmieniają. Gdy wielomian $x^{100}+1$, zamieniamy na $x^{10}+1$, wystarczy, jedyne co musimy zrobić, to usunąć wartość dla klucza równego 100, a dodać klucz równy 10, z wartością 1.

W porównaniu z PolynomialVector, klasa ta wydaje się lepsza dla rzadkich struktur wielomianów, a gorsza dla gęstych. Przeanalizujmy krótko, dlaczego tak podpowiada intuicja. Niepodważalne jest, że pojedynczy dostęp do dowolnego elementu, w przypadku tablicy jest szybszy, niż dla jakiejkolwiek innej struktury. Dzieje się tak, ponieważ na podstawie miejsca początku tablicy, dla dowolnego jej elementu, na podstawie indeksu, z góry znamy jego dokładny adres. W przypadku wszystkich kontenerów danych, których kolejne elementy, nie są położone w pamięci obok siebie, najpierw musimy je znaleźć, tj. uzyskać ich adres.

Rozważmy strukturę danych, jaką jest drzewo. Jego elementy są posortowane, ale bezpośredni dostęp mamy tylko do korzenia. Do innych elementów, można dostać się w sposób, wyłącznie pośredni. Niezależnie od rodzaju drzewa, jaki wybierzemy, nie jesteśmy w stanie przekroczyć pewnych charakterystycznych dla niego wartości. W przypadku drzewa, średnia złożoność, znalezienia wybranej wartości jest logarytmiczna. Mowa tu, o najkorzystniejszym wariancie, czyli przypadku, gdy drzewo to jest zrównoważone, tzn. takie, w którym, odległość do jego liści jest różniąca się maksymalnie o jeden i bliska wartości logarytmu z rozmiaru drzewa. Dodatkowo każde doda-

nie, modyfikacja i usunięcie dowolnego elementu, zaczyna się od jego znalezienia. Oznacza to, że złożoność logarytmiczna jest w takim przypadku, wartością graniczną.

Wydaje się więc, że aby uzyskać w przypadku takiej struktury, wydajność lepszą niż dla tablicy, liczba ich elementów musi się odpowiednio różnić. Potrzebuje ona mieć odpowiedni zapas liczby operacji, by ich czas, pomimo długości każdej z nich, był łącznie mniejszy.

3.3 Główne funkcje

3.3.1 Metody klasy Parser

Unifikacja wejściowego łańcucha znaków

```
inline string Parser::UniformInputString(string s)
```

Zadaniem funkcji jest ujednolicenie wielomianu podanego na wejściu, w taki sposób, by był jednoznaczny i łatwy do dalszego przetworzenia. Jako argument przyjmuje ona pojedynczy obiekt klasy string. Po dokonaniu odpowiednich operacji, jako rezultat, zwraca również typ string. Użycie jej ma na celu ułatwienie przetwarzania w kolejnym etapie, w którym na podstawie podanego ciągu znaków, tworzony będzie obiekt wielomianu. Głównym zadaniem funkcji, jest weryfikacja, czy ciąg znaków podanych na wejściu reprezentuje poprawny składniowo wielomian. Sprawdzana jest liczba nawiasów otwierających i zamykających oraz fakt, czy w każdym miejscu wyrażenia liczba otwartych nawiasów jest nie mniejsza niż liczba nawiasów zamkniętych. Dodatkowo sprawdzane jest, czy na sąsiednich miejscach nie występują dwa operatory. Poprawne wyrażenie nigdy nie kończy się operatorem, a zaczynać może się tylko minusem, literałem, cyfrą lub nawiasem otwierającym. W czasie unifikacji wyrażenia, ignorowane są wszystkie występujące w nim białe znaki. Warto zaznaczyć, że dopuszczalny jest tylko jeden znak, reprezentujący zmienną wielomianu. Nie ma ograniczeń co do wartości tego znaku, może być to znak 'a', 'x' lub jakikolwiek inny, ale powinniśmy zadbać, by w całym wyrażeniu występował on w tej samej postaci. Sporym ułatwieniem w interfejsie jest brak konieczności wpisywania operatorów mnożenia ('*') i potęgowania ('^') w oczywistych miejscach. Z punktu widzenia aplikacji, wyrażenia 4x oraz 4*x są identyczne. Podobnie jest w przypadku x3 oraz x^3. Funkcja analizuje wyrażenie i zwraca je w odpowiedniej postaci. Dla przykładu wyrażenia x3+2x2+3x+1, x^3 +2x^2 + 3x+1 zostaną zamienione w x^3+2*x^2+3*x+1.

Konwersja łańcucha znaków do postaci wielomianu

```
inline Polynomial&
 Parser::ConvertToPolynomial(string inputS, int type)
```

Funkcja ma na celu stworzenie wielomianu na podstawie wartości znakowej i zwrócenie odpowiedniej referencji. W zależności od parametru type, wielomian reprezentowany jest przez jeden z dwóch rodzajów struktur. Pierwszą z nich jest wektor – począwszy od wyrazu stojącego przy najwyższej potędze, reprezentowane są wszystkie współczynniki, także zerowe. Drugim jest mapa – która posiada informacje tylko o niezerowych współczynnikach.

Na początku funkcji wykonywana jest metoda UniformInputString. Jeżeli zwróci ona pustą wartość, to funkcja przerywa swoje działanie, zwracając pusty obiekt wielomianu. W przeciwnym wypadku następuje iteracja po kolejnych znakach łańcucha wejściowego. Ze względu na kolejność wykonywania działań, w funkcji występują trzy obiekty wielomianów. Pierwszy z nich posiada informacje o aktualnie przetwarzanym fragmencie wejścia. Drugi mówi o wartości fragmentu wielomianu, który wystąpił przed znakiem mnożenia lub dzielenia. Stanowi on, obok aktualnego wielomianu, drugi operand dla wskazanej operacji. Z kolei, trzeci jest analogiczny i mówi o wartości drugiego operandu dla dodawania i odejmowania. Wszystkie te działania zostają dokonywane dopiero w momencie, gdy natrafimy na dany operator.

W przypadku, gdy mamy do czynienia ze znakiem nawiasu otwierającego, zostaje wyszukany odpowiedni nawias zamykający, a na wewnętrznym zakresie jest wywoływana rekurencyjnie ta sama funkcja. Zwracana wartość zapisywana jest jako aktualny element, a wskaźnik zostaje przesunięty na kolejny. Gdy aktualnie przetwarzanym znakiem jest potęgowanie, zachowanie funkcji jest analogiczne. Wielomian zostaje podniesiony do odpowiedniej potęgi, otrzymana wartość zapisana, a wskaźnik ustawiony na kolejny element, występujący po wykładniku.

W przypadku pozostałych operatorów działanie funkcji jest nieco odmienne. Spowodowane jest to faktem, że trafiając na dany operator, nie znamy jeszcze jego obu argumentów, a jedynie pierwszy z nich. Drugi jest nieznany, a dodatkowo sam w sobie może zawierać operatory z wyższym priorytetem, np. w przypadku wielomianu $x + 3 * 2$ potrzebujemy najpierw obliczyć wartość wyrażenia $3 * 2$ i dopiero wówczas uzyskany wynik zsumować z pierwszym operandem, równym x. Fakt ten, dopiero w momencie natrafienia na dany operator, narzuca wykonywanie poprzedniego działania. Uwaga ta tyczy się jednak tylko natrafienia na znaki dodawania, odejmowania, mnożenia i dzielenia, ponieważ, jak zostało wspomniane wcześniej, operator potęgowania i nawiasy posługują się osobnym algorytmem. Zachowanie to, można porównać do działania stosu, na którym mogą leżeć tylko dwa argumenty. Pierwszym z nich jest operacja dodawania, a drugim mnożenia. Za pomocą nich można przedstawić także odejmowanie i dzielenie, zatem funkcje te nie będą przeze mnie rozpatrywane osobno. Operator mnożenia jest traktowany jako działanie z wyższym priorytetem, a dodawania z niższym. Oznacza to, że aby wykonać sumowanie, należy zadbać o to by ewentualna operacja mnożenia została wykonana wcześniej. Tak więc, w momencie natrafienia na znak mnożenia lub dzielenia, wystarczy przemnożyć lub podzielić przez siebie dwa operandy. W momencie dodawania, sytuacja jest bardziej skomplikowana. Zostaje wykonane sprawdzenie, czy odłożony został operator mnożenia. Jeżeli tak, to wartość aktualnego elementu zostaje przemnożona przez zapisany argument, w przeciwnym razie pozostaje bez zmian. Następnie, wynik operacji zostaje dodany lub odjęty od zapisanego argumentu. W momencie, gdy zakończymy iterowanie po całym łańcuchu wejściowym, należy zadbać o ewentualne wykonanie mnożenia oraz dzielenia, czyniąc to analogicznie jak w momencie rozpatrywania znaku dodawania lub odejmowania.

3.3.2 Metody klasy Number

Sortowanie liczb typu mpq_t

```
inline vector<Number> SortNumbers(vector<Number>v)
```

Zadaniem tej metody jest posortowanie liczb, z biblioteki mpir, w kolejności niemalejącej. Zarówno wejściem, jak i wyjściem funkcji jest wektor obiektów klasy Number. Metoda ta jest używana w celu sortowania tablicy, o niewielkiej liczbie elementów. Oznacza to, że w takim przypadku, bardzo dobrze sprawdzi się nieskomplikowany algorytm sortujący, np. sortowanie bąbelkowe. Z uwagi na mały zestaw danych, rzędu kilkudziesięciu elementów, kwadratowa złożoność czasowa, będzie akceptowalna, gdy weźmiemy pod uwagę bardzo niewielką liczbę koniecznych operacji, w jednym obiegu pętli. Metoda skorzysta z operatora porównania i metody zamiany elementów klasy Number, bazujących na funkcjach mpq_cmp oraz mpq_swap, z biblioteki mpir.

Porównywanie wektorów liczbowych

```
inline int VectorsAreEqual(vector<Number>v1, vector<Number>v2)
```

Głównym celem metody jest sprawdzenie, czy dwa wektory liczbowe są sobie równe, tzn. czy zawierają wszystkie te same elementy, w dowolnej kolejności. Funkcja przyjmuje, jako argumenty, dwa wektory, a zwraca liczbę typu int. Jest ona równa 0, gdy wektory są różne oraz 1, gdy są równe. W pierwszej kolejności porównywana jest liczba elementów obu wektorów. Gdy jest ona różna, wówczas mamy pewność, że wektory są różne. Gdy liczba elementów jest równa, elementy w obu tablicach zostają posortowane, przy pomocy funkcji SortNumbers. Następnie porównywane są kolejne elementy wektorów. Wektory uznajemy za inne, gdy chociaż jedna para jest różna.

Wyszukiwanie danej liczby w tablicy

```
inline int Number::IsInVector(vector<Number> v)
```

Metoda stwierdza, czy obiekt, na którym będzie wykonywana, jest elementem wektora podanego jako argument funkcji. Zwraca ona -1, gdy tablica nie zawiera danej wartości liczbowej. W przeciwnym wypadku, rezultatem funkcji jest indeks w podanej tablicy, przy czym pierwszy jej element ma indeks równy 0. Z uwagi, na to, że dany wektor nie zawsze jest posortowany, nie możemy skorzystać z algorytmu wyszukiwania binarnego, posiadającego logarytmiczną złożoność, ze względu na liczbę jego elementów. By z niego skorzystać, musielibyśmy wykonać sortowanie, a także zapamiętywać informację o indeksie danej liczby, w wektorze inicjalnym. Najszybszy algorytm sortowania, wykonywany na jednym wątku, ma złożoność liniowo-logarytmiczną. Jego użycie jest wiec zupełnie nieopłacalne, jeżeli weźmiemy pod uwagę, że przejrzenie wszystkich liczb tablicy, ma dokładnie taką samą złożoność. Dodatkowo zastosowany algorytm ma własność stopu, tzn. gdy natrafi na element, którego szukamy, kończy swoje działania, zwracając odpowiedni indeks.

Zwracanie wartości liczby w postaci łańcucha znaków

```
inline string Number::ToString()
```

Funkcja ma celu przekształcenie danej liczby, standardowo podawanej jako ułamek, w postaci – licznik i mianownik, na liczbę, zapisaną w postaci dziesiętnej. By to osiągnąć, potrzebna jest informacja o wymaganej precyzji obliczeń. Na początku liczba zostaje przyrównana do zera, gdy jest mniejsza, jako pierwszy znak wyniku, zostaje ustawiony minus, a dalej liczba jest zawsze rozpatrywana jako nieujemna. Na początku zostaje obliczony iloczyn licznika i liczby równej 10^x,

gdzie x – precyzja, wyrażona w ważnych cyfrach po przecinku. Następnie obliczany jest całkowitoliczbowy wynik dzielenia. Do otrzymanego w ten sposób rezultatu, wystarczy już tylko dodać separator, oddzielający część całkowitą i ułamkową w odpowiednim miejscu. Przypada on w takim miejscu, by po stronie ułamkowej znajdowała się dokładnie żądana liczba cyfr. Na koniec, należy jeszcze zadbać, by otrzymany wynik był dobrze sformatowany, trzeba więc usunąć, nieznaczące zera w części ułamkowej. Dodatkowo potrzeba pamiętać, że po wykonaniu takiej operacji, może zdarzyć się, że część ułamkowa nie zawiera już żadnych cyfr. Wówczas należy nie zapomnieć, o usunięciu znaku separatora, tak by dla powstałej liczby całkowitej ostatnim w kolejności znakiem była cyfra jedności.

3.3.3 Metody klasy Polynomial

Największy wspólny dzielnik wielomianów

```
inline Polynomial&
 Polynomial::Nwd(Polynomial& p1, Polynomial& p2)
```

Jest to funkcja obliczająca największy wspólny dzielnik dwóch wielomianów. Wynik jest zwracany przy pomocy referencji do uzyskanego wielomianu. Na początku obliczana jest wartość reszty z dzielenia ilorazu podanych wielomianów. Funkcja wołana jest w sposób rekurencyjny. Kolejnymi argumentami funkcji dla wielomianów p_1 i p_2 są wielomian p_2 oraz reszta z dzielenia wielomianu p_1 przez wielomian p_2. Funkcja wołana jest rekurencyjnie, pod warunkiem, że otrzymana reszta, nie jest wielomianem stopnia zerowego. W takim przypadku zwracana jest wartość p_2, jeżeli dzieli ona bez reszty wielomian p_1 lub wielomian $W(x) = 1$, w przeciwnym wypadku. Warto zauważyć, że z każdym wywołaniem funkcji, stopień wielomianów, które są jej argumentami, maleje. Oznacza to, że dla wielomianów stopnia n i m, maksymalna liczba rekurencyjnych wywołań funkcji jest równa $\min(n, m) + 1$.

Eliminacja pierwiastków wielokrotnych wielomianu

```
inline Polynomial&
Polynomial::PolynomialAfterEliminationOfMultipleRoots()
```

Funkcja dla danego wielomianu, dokonuje eliminacji pierwiastków wielokrotnych i zwraca otrzymany w ten sposób wielomian. Na początku funkcji, obliczana jest pochodna wielomianu, a następnie dla tych wielomianów, znajdowany jest największy wspólny dzielnik. Na otrzymywanych wielomianach dokonywana jest operacja normalizacji, tzn. podzielenie wszystkich jej współczynników, przez wartość, równą współczynnikowi, stojącemu przy najwyższej potędze. W ten sposób otrzymamy wielomian, o tych samych pierwiastkach, ze współczynnikiem, przy najwyższej potędze równym 1. Pozwala to na dokonanie łatwiejszych, dzięki czemu często szybszych, obliczeń, na danych wielomianach. W drugiej części funkcji, dokonywana jest eliminacja pierwiastków wielokrotnych, poprzez podzielenie wielomianu W, przez wielomian równy $NWD(W, W')$. Otrzymany wynik jest rezultatem funkcji. Zarówno dzielna, jak i dzielnik, są wielomianami, dla których dokonaliśmy już normalizacji, wiadomo więc, że wynikowa wartość, jest już znormalizowana.

Algorytm wyznaczania kolejnych przedziałów wyszukiwania

```
inline Number
Polynomial::NextNumberFromRange(Number a, Number b)
```

Funkcja zwraca wybraną liczbę z przedziału (a, b). Algorytm funkcji działa tak, by zoptymalizować wybieranie przedziałów, w których sprawdzane będzie istnienie pierwiastków. Ma to na celu jak najszybsze zawężenie przedziału, w którym znajduje się szukany pierwiastek. Zakładamy, że długość przedziału jest niezerowa, a liczby a i b reprezentują odpowiednio – lewy i prawy kraniec przedziału. Z tego założenia wynika, że $a < b$ i takich parametrów spodziewa się funkcja. W celu optymalizacji, wewnątrz funkcji nie jest sprawdzane, czy przedstawiony warunek jest spełniony. Poniżej przedstawiona została wartość zwracana przez funkcję, w zależności od otrzymanych na wejściu parametrów.

$$f(a,b) = \begin{cases} -2,\ dla\ a = -\infty \wedge b = -1 \\ -1,\ dla\ a = -\infty \wedge b = 0 \\ 0,\ dla\ a \cdot b < 0 \\ 1,\ dla\ a = 0 \wedge b = +\infty \\ 2,\ dla\ a = 1 \wedge b = +\infty \\ a \cdot |a|,\ dla\ a\ \in (0,1) \cap (1,+\infty) \cap \{-1\} \wedge b = -\infty \\ b \cdot |b|,\ dla\ a = -\infty\ \wedge\ b \in (-\infty,-1) \cap (-1,0) \cap \{1\} \\ \frac{a+b}{2}, \text{w pozostałych przypadkach} \end{cases}$$

Jak widać, funkcja w pierwszej kolejności stara się dokonać takiego podziału, by w obu podprzedziałach znajdowały się wartości tego samego znaku. Kolejnymi wartościami, o których należy wspomnieć są liczby -1 oraz 1. Dzieje się tak z uwagi na to, żeby zoptymalizować znajdowanie pierwiastków, zarówno takich, o niewielkiej wartości bezwzględnej, bliskich zeru, jak i, bardzo dużych. Wówczas poprzez obliczanie kwadratu aktualnej wartości, z zachowaniem jej dotychczasowego znaku, jesteśmy w stanie, w niewielkiej liczbie iteracji, maksymalnie zawęzić szukany przedział. Warto zauważyć, że dla liczb mniejszych od 1, posiadających, jako drugi kraniec przedziału 0, w tempie wykładniczym zbliżamy się do 0. Natomiast dla liczb większych od 1, gdy drugim krańcem przedziału jest $+\infty$, w tym samym tempie się od niego oddalamy. Przeanalizujmy jak powstają kolejne podziały w dwóch przypadkach, gdy przedział jest równy $(-\infty, \infty)$, a szukany pierwiastek to -0.1 oraz gdy przedziałem jest $(-1, 30)$, a pierwiastkiem 5. W celu wygody i przejrzystości prezentowanych obliczeń, w przykładzie przyjmijmy, że szukamy pierwiastka z dokładnością 0.1. Oznacza to, że w momencie, gdy mamy przedział długości nie większej niż 0.2, o którym wiemy, że znajduje się w nim pierwiastek, to jesteśmy w stanie podać jego wartość z żądaną precyzją, wskazując dokładnie środek danego przedziału.

Przykład 11

$a = -\infty,\ b = +\infty\ \Rightarrow\ c = 0$
$a = -\infty,\ b = 0\ \Rightarrow\ c = -1$
$a = -1,\ b = 0\ \Rightarrow\ c = -0.5$

$a = -0.5,\ b = 0\ \Rightarrow\ c = -0.25$
$a = -0.25,\ b = 0\ \Rightarrow\ c = -0.0625$

Jak widać dla tego specyficznego przypadku, algorytm już w 5 krokach ustalił wartość poszukiwanej liczby. Długość przedziału jest równa $-0.0625 - (-0.25) = 0.1875 < 0.2$, zatem podając środek otrzymanego przedziału, równy $\frac{-0.0625+(-0.25)}{2} = \frac{-0.3125}{2} = 0.15625$, mamy pewność, że będzie wskazywał on badany pierwiastek, z wymaganą precyzją. Teraz spójrzmy na drugi scenariusz.

Przykład 12 $a = -1,\ b = 30\ \Rightarrow\ c = 0$
$a = 0,\ b = 30\ \Rightarrow\ c = 15$
$a = 0,\ b = 15\ \Rightarrow\ c = -7.5$
$a = 0,\ b = 7.5\ \Rightarrow\ c = 3.75$
$a = 3.75,\ b = 7.5\ \Rightarrow\ c = 5.625$
$a = 3.75,\ b = 5.625\ \Rightarrow\ c = 4,6875$
$a = 4.6875,\ b = 5.625\ \Rightarrow\ c = 5,15625$
$a = 4.6875,\ b = 5.15625\ \Rightarrow\ c = 4.921875$

W drugim przypadku, funkcja zakończyła działanie po 8 iteracjach. Możemy zauważyć, że liczba kroków jest zależna bezpośrednio od żądanej precyzji. Podając jako rozwiązanie kolejne środki przedziału, jesteśmy w stanie w czasie logarytmicznym dla badanego przedzialu, znaleźć leżący w nim pierwiastek. Należy zwrócić uwagę, że w niekorzystnym przypadku, kolejne przybliżenia niekoniecznie muszą być coraz bliższe szukanemu rozwiązaniu. Tak byłoby, gdy pierwiastkiem w powyższym przykładzie była liczba 5.15. Wówczas pomimo bycia w bliskim jego otoczeniu, kontynuowalibyśmy pracę algorytmu, a ten znalazłby szukane rozwiązanie, dopiero kilka iteracji później. Jest to niewątpliwy minus zastosowanego algorytmu, jednak ciężki do wyeliminowania.

Liczba pierwiastków wielomianu w badanym przedziale

```
inline int Polynomial::NumberOfRoots(Number a, Number b)
```

Funkcja zwraca liczbę pierwiastków w zadanym przedziale. Jest ona obliczana na podstawie liczby zmian znaku na krańcach przedziałów i równa liczbie takich zmian na prawym z nich, pomniejszona o ich wartość na lewym.

Znajdowanie pierwiastków wielomianu w badanym przedziale

```
inline vector<Number> Polynomial::FindRoots(Number a, Number b)
```

Jest to najważniejsza funkcja całego projektu, ponieważ wewnątrz niej znajduje się cała logika projektu. Zwraca ona tablicę, z wartościami kolejnych pierwiastków wielomianu. Na początku funkcji sprawdzana jest liczba pierwiastków w badanym przedziale. Liczba ich nie może być ujemna, a kiedy równa jest 0, funkcja kończy swoje działanie i zwraca pusty wektor.

Zgodnie z twierdzeniem, liczba pierwiastków w przedziale, wlicza także ewentualny pierwiastek na prawym krańcu przedziału. Gdy funkcja stwierdzi istnienie tego pierwiastka, zostaje on dodany

do wektora wyjściowego. Jeżeli liczba pierwiastków w przedziale była równa 1, wiemy już, że był to jedyny pierwiastek w danym przedziale i możemy zwrócić jednoelementowy wektor, z wartościami pierwiastków.

Następnie, na podstawie funkcji NextNumberFromRange, przedział zostaje podzielony na dwa mniejsze. W otrzymanych przedziałach zostaje obliczona liczba pierwiastków. Jeżeli jest ona dodatnia, to następuje rekurencyjne wywołanie funkcji, z argumentami, będącymi granicami danego podprzedziału. Wywołana rekurencyjnie metoda, po wykonaniu, zwróci wynik, który zostanie przetworzony przez funkcje wywołującą.

3.3.4 Metody czysto wirtualne klasy Polynonial – porównanie działania metod klas PolynomialMap i PolynomialVector

Ustawianie wartości wyrazu wielomianu

```
inline void
    PolynomialMap::SetNumberValue(int power, Number number)
inline void
    PolynomialVector::SetNumberValue(int power, Number number)
```

Głównym zadaniem funkcji jest ustawienie wartości podanego współczynnika na daną wartość liczbową. W przypadku mapy, konieczna jest weryfikacja, czy liczba jest równa zero. Jeżeli tak, to następuje sprawdzenie, czy w mapie występuje już klucz, równy podanej potędzę. W takim przypadku następuje jego usunięcie, a gdy klucz nie istnieje, jej opuszczenie, bez jakichkolwiek dodatkowych działań. Jeżeli wartość liczbowa jest niezerowa, to sprawdzenie jest analogiczne. W pierwszym przypadku następuje wówczas nadpisanie wartości o danym kluczu, a w drugim wstawienie do mapy pary – klucz (potęga), wartość (liczba).

W przypadku klasy PolynomialVector metoda została nieco bardziej rozbudowana. Na początku następuje sprawdzenie, czy dany stopień potęgi jest wyższy od aktualnego stopnia wielomianu. Jeżeli tak, to następuje sprawdzenie, czy dana wartość jest zerowa. Wówczas funkcja kończy swoje działanie, a w przeciwnym wypadku rozpoczyna operację wstawiania do wektora żądanej pary – potęgi i wartości współczynnika. Gdy stopień wielomianu zwiększa się o 1, wystarczy, po prostu, na kolejnym miejscu w tablicy wstawić daną wartość. W przeciwnym zaś przypadku, musimy zadbać o to, by na miejscach wszystkich współczynników, reprezentujących ich wartości dla kolejnych potęg, począwszy od dotychczasowego stopnia wielomianu, powiększonego o 1, aż do jego nowego stopnia, pomniejszonego o 1, zostały wstawione zera. Dopiero wówczas na kolejnym miejscu w tablicy może zostać wstawiona wartość dla odpowiedniej potęgi.

Gdy podana w argumencie funkcji potęga, jest niewiększa od stopnia wielomianu, następuje uaktualnienie wartości na odpowiedniej pozycji w tablicy. Konieczne jest wówczas dodatkowe sprawdzenie. Jeżeli podana para (potęga, wartość) współczynika wskazują na zerowy współczynnik dla potęgi, równej stopniowi danego wielomianu, niezbędne jest przesunięcie końca tablicy o jedno miejsce. Następnie wykonywane są sprawdzenia kolejnych elementów funkcji, począwszy od końca. W przypadku, gdy mamy do czynienia z zerami, następują kolejne korekty granicy tablicy. Podana

operacja trwa tak, dopóki nie zostaną przeanalizowane wszystkie elementy lub nie natrafimy na dowolną wartość różną od zera.

Warto zauważyć, że w przypadku obu klas, funkcja ma wpływ na stopień wielomianu. Dodatkowo, operacja dokładania i zdejmowania kolejnych argumentów, w przypadku wektora nie została zoptymalizowana. Możliwe, że w sytucji konieczności, wielokrotnej zmiany jej rozmiarów, korzystniejsza byłaby pojedyncza operacja, zwłaszcza w przypadku konieczności usuwania kolejnych zerowych współczynników. Należałoby wówczas, najpierw przeanalizować kolejne ich wartości i zliczyć wszystkie zera znaczące, a następnie pojedynczą operacją, odpowiednio zmanipulować wskaźnik na ostatni element.

Stwierdzanie czy wielomian jest wielomianem zerowym

```
inline bool PolynomialMap::IsZero()
inline bool PolynomialVector::IsZero()
```

Funkcja ma na celu przeanalizowanie struktury wielomianu i stwierdzenie, czy dany wielomian jest wielomianem zerowym. Implementacja metody w obu klasach jest analogiczna. Następuje sprawdzenie, czy wielomian posiada wyłącznie zerowe współczynniki. Zgodnie z założeniami, w mapie przetrzymywane są tylko niezerowe wartości wyrazów wielomianu, a w wektorze, przechowywane są wszystkie współczynnik, aż do ostatniego niezerowego współczynnika, stojacego przy najwyższej potędze. Na tej podstawie jesteśmy w stanie stwierdzić, że funkcja zwraca prawdę, tylko, jeżeli mapa, bądź wektor są puste.

Obliczanie stopnia wielomianu

```
inline int PolynomialMap::PolynomialDegree()
inline int PolynomialVector::PolynomialDegree()
```

Zadaniem tej metody jest obliczenie stopnia danego wielomianu. Złożoność funkcji jest zależna od wybranej klasy. W przypadku PolynomialMap konieczna jest analiza jej kolejnych elementów i stwierdzenie, jaka jest największa, występująca w niej potęga. W przypadku PolynomialVector złożonośc funkcji jest stała, gdyż stopień wielomianu jest równy rozmiarowi tablicy, pomniejszonego o jeden. Dla obu klas, gdy mamy do czynienia z wielomianem zerowym, zwracaną wartością jest -1.

Zwracanie wartości wielomianu w postaci łańcucha znaków

```
inline string PolynomialMap::ToString()
inline string PolynomialVector::ToString()
```

Celem funkcji jest zwrócenie wartości wielomianu w postaci zmiennej znakowej. Jej rezultat wygląda analogicznie, jak wynik metody UniformInputString w klasie Parser. Na podstawie odpowiednich par – potęgi i wartości współczynnika, zostaje stworzony obiekt typu string. Na podstawie kolejnych liczb, przed kolejnymi wyrazami, zostają dostawione odpowiednio znaki plus, dla wartości większych od 0 oraz minus, dla wartości mniejszych. Wyjątkiem jest pierwszy wyraz, w przypadku którego ewentualny znak minus, jest ignorowany. Pomijane są także zbędne elementy, takie jak

"1*" dla współczynników, stojących przy potęgach dodatnich oraz "*x^0" "^1" dla odpowiednio – zerowej i pierwszej potęgi.

Chociaż struktury różnią się reprezentacją w pamięci, to obie zwracają identycznie sformatowany rezultat. Uznałem bowiem, że w przypadku tablicy, przy przyjętym formatowaniu, wypisywanie zerowych współczynników jest zupełnie zbędne i tylko uczyniłoby wynik mniej czytelnym. Opcja wypisywania tablicy z kolejnymi współczynnikami została także odrzucona z podobnych powodów. Sytuacja taka byłaby bardzo niekorzystna dla użytkownika. Zwłaszcza dla rzadkich wielomianów wysokich stopni, ich reprezentacja uległaby znacznej zmianie, a ręczne znalezienie niezerowych współczyników okazałoby się praktycznie niewykonalne.

Dzielenie wielomianów

```
inline pair<Polynomial&, Polynomial&>
    PolynomialMap::DividePolynomials(Polynomial& p1, Polynomial& p2)
inline pair<Polynomial&, Polynomial&>
    PolynomialVector::DividePolynomials(Polynomial& p1, Polynomial& p2)
```

Metoda dla podanych wielomianów oblicza ich iloraz oraz resztę z dzielenia i zwraca je jako parę obiektów. Na początku, funkcja oblicza wartość pierwszego wyrazu wielomianu, który zostanie zwrócony, jako iloraz. Jest on równy wynikowi dzielenia wyrazów stojących przy najwyższych potęgach wielomianów – dzielnej i dzielnika. Kolejne wyrazy drugiego z nich są mnożone przez otrzymaną wartość, a wynik jest odejmowany od dzielnej. W rezultacie tego działania, współczynnik przy najwyższej potędze staje się równy zero i zostaje zredukowany. Daje to gwarancję, że nowy stopień wielomianu, będzie przynajmniej o jeden mniejszy, niż stopień danego wielomianu. Funkcja kończy swoje działanie, gdy dokona powyższej operacji dla wszystkich wyrazów dzielnej. Wówczas aktualnie przetwarzany wielomian jest zwracany jako obliczona reszta. Warto zauważyć, że stopień tego wielomianu jest mniejszy od stopnia dzielnika i co najmniej o 2 mniejszy od stopnia dzielnej.

Obliczanie pochodnej wielomianu

```
inline Polynomial& PolynomialMap::Derivative()
inline Polynomial& PolynomialVector::Derivative()
```

Celem funkcji jest obliczenie pochodnej danego wielomianu. Jej wartość jest obliczana tylko w przypadku, gdy wielomian jest niezerowy. Dla danego wielomianu $W(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$, pochodna wielomianu wyraża wzorem: $W'(x) = na_0x^{n-1} + (n-1)a_1x^{n-2} + ... + 2a_{n-2}x + a_{n-1}$. W przypadku klasy PolynomialMap wymagane jest przejrzenie tylko niezerowych współczynników wielomianu, zatem operacja ta jest liniowa, względem ich liczb. Z kolei, dla typu PolynomialVector konieczne jest sprawdzenie wartości wszystkich współczynników, przy czym tylko te niezerowe mają wpływ na wartość pochodnej.

Obliczanie wyrazów ciagu Sturma

```
inline vector<PolynomialMap> PolynomialMap::GetSturm()
inline vector<PolynomialVector> PolynomialVector::GetSturm()
```

Funkcja ma na celu zwrócenie wektora wielomianów, w którym wartościami będą kolejne wyrazy ciągu Sturma. Na początku, przy pomocy funkcji Derivative obliczana jest pochodna wielomianu. Gdy jest ona wielomianem zerowym, zostaje zwrócony ciąg Sturma, zawierający tylko jeden element – wielomian, dla którego ciąg ten jest obliczany. Wiadomo wówczas, że liczba zmian znaków dla takie ciagu, niezależnie od wybranego punktu, będzie równa zero, co oznacza, że dany wielomian nie zawiera pierwiastków rzeczywistych.

Kolejnym krokiem jest obliczenie reszty z dzielenia wielomianu, przez jego pochodną. Jeżeli jest ona wielomianem zerowym, to zwracany jest dwuelementowy ciąg Sturma – wielomian i jego pochodna. W przeciwnym razie, wartość do niej przeciwna stanowi kolejny element ciągu Sturma. Podobnie jak w przypadku pierwszej reszty z dzielenia, także kolejne wyrazy są obliczane jako iloraz dwóch poprzednich wyrazów ciągu Sturma. Dzieje się to tak długo, dopóki otrzymany wielomian jest wielomianem zerowym. Warto zauważyć, że maksymalna liczba wyrazów ciągu Sturma jest równa stopniowi wielomianu, powiększonemu o jeden. Wówczas stopnie kolejnych wyrazów ciągu Sturma są równe kolejnym potęgom, od stopnia wielomianu, do zera.

Liczba zmian znaków w danym punkcie

```
inline int PolynomialMap::NumberOfChangesSign(Number a)
inline int PolynomialVector::NumberOfChangesSign(Number a)
```

Funkcja na postawie ciągu Sturma i metody GetSturm, oblicza liczbę zmian znaków dla podanego punktu. Porównuje ona wartości dwóch kolejnych wyrazów ciagu Sturma. Jeżeli ich iloczyn jest ujemny, następuje zwiększenie licznika o jeden. W przypadku wartości zerowych, element taki jest pomijany, tzn. ciąg jest rozpatrywany, tak jakby go nie zawierał. Warto zauważyć, że dla odpowiednio dużych wartości bezwzględnych liczby a, w tym wartości niewłaściwych $-\infty$ i $+\infty$, liczba zmian znaków, zależy wyłącznie od współczynników stojących, przy najwyższych potęgach kolejnych wielomianów. Fakt ten powoduje możliwą optymalizację działania funkcji, w zależności od otrzymanego parametru. Najbardziej czasochłonna operacja, jaką jest dzielenie wielomianów, i tak musi zostać wykonana, by znaleźć wszystkie wyrazy ciągu Sturma, ale zysk z takiej optymalizacji, może być zauważalny i znacząco poprawić wydajność funkcji.

3.4 Instalacja aplikacji

Stopień trudności instalacji aplikacji zależny jest od typu procesora, na którym będzie uruchomiony nasz program oraz od tego, w jaki sposób chcemy jej używać. Należy mieć świadomość, że cały poniższy opis instalacji dotyczy przypadku 64-bitowego systemu Windows, jako platformy pod którą wymagane jest działanie aplikacji.

3.4.1 Szybka instalacja

Pliki binarne aplikacji zostały skompilowane przy użyciu kompilatora wbudowanego w Microsoft Visual Studio 2015. Sprawia to, że aplikacja korzysta z dodatkowych bibliotek dynamicznych, które musimy posiadać na komputerze, na którym uruchamiamy aplikacje. Gdy tego nie zrobimy,

w trakcie uruchomienia dostaniemy komunikat o braku wymaganego pliku. Możemy dodać go, nie istalując powyższego środowiska, ale musimy być świadomi, że przy kolejnej próbie uruchomienia dostaniemy bliźniaczy komunikat dla innego pliku. Tak, więc rekomendowanym sposobem jest instalacja Microsoft Vistual Studio 2015, który zainstaluje wszystkie potrzebne biblioteki.

Drugą rzeczą, którą należy zapewnić jest dynamiczna biblioteka mpir – w postaci pliku mpir.dll. Jeżeli instalujemy aplikację na komputerze z procesorem Intel i7 plik ten znajduje się w folderze z aplikacją. W przeciwnym razie jesteśmy zmuszenie skompilować źrodła biblioteki mpir, a powstały plik "mpir.dll" dodać do folderu z aplikacją lub pod ogólną lokalizacją dla wszystkich bibliotek systemu Windows.

3.4.2 Kompilacja biblioteki mpir

Gdy chcemy uruchomić program na innym procesorze niż Intel i7 potrzebujemy dołączyć do niego dynamiczną bibliotekę mpir. Aby to zrobić, musimy dokonać kompilacji jej źrodeł dla wybranego procesora, a następnie skopiować uzyskany plik typu dll do odpowiedniego folderu.

Zaczynamy od uzyskania źrodeł biblioteki ze strony "http://mpir.org/". Następnie otwieramy pobraną solucję w programie Miscrosoft Visual Studio 2015. Ważne jest to, by w ustawieniach zmienić opcję kompilacji "Debug" na "Release" i wybrać system 64-bitowy. Następnie wybieramy projekt przeznaczony na interesującą nas platformę i budujemy go.

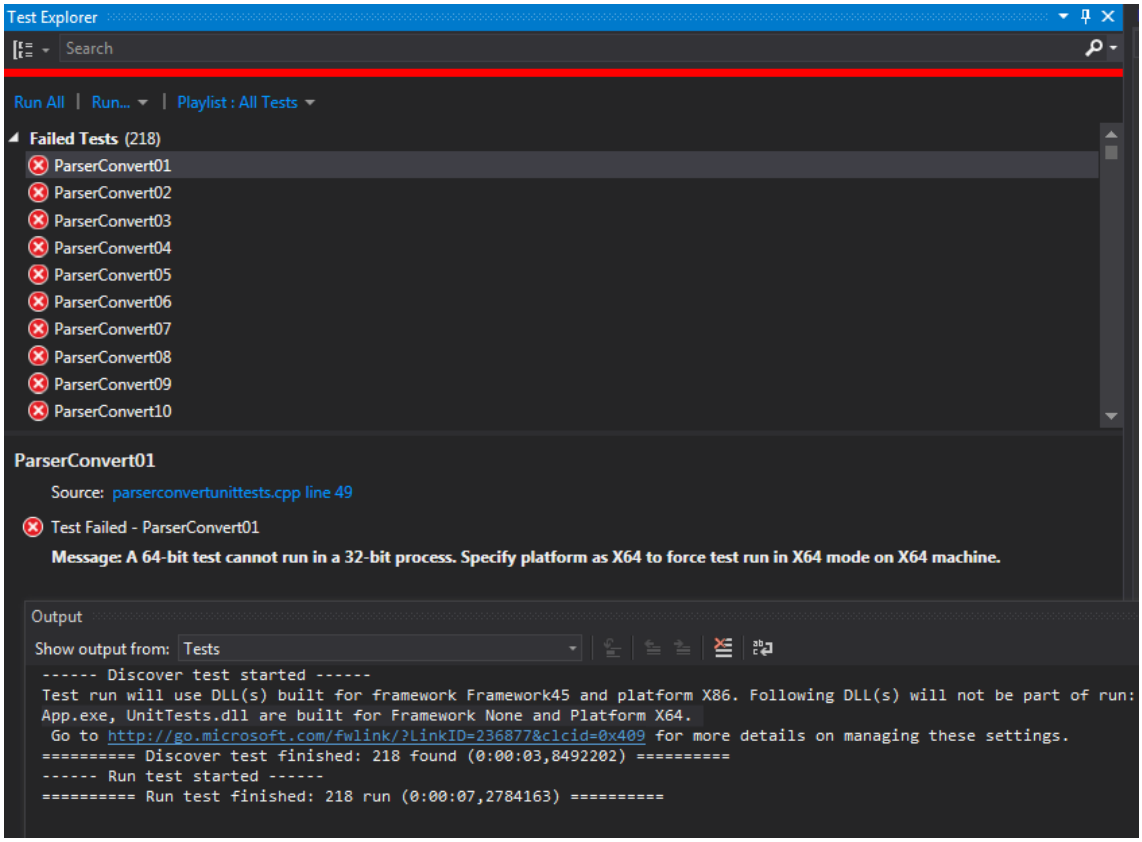
Jeżeli proces budowania zakończył się powodzeniem, dysponujemy już potrzebnym plikiem. Znajduje się on w podfolderze projektu, który budowaliśmy, w katalogu "x64/Release".

3.4.3 Kompilacja solucji

Jeżeli interesuje nas zmiana plików źródłowych aplikacji, musimy mieć dostęp zarówno do plików źródłowych biblioteki mpir, jak i tych, które powstały w procesie budowania. Dlatego pierwszym krokiem powinno być w tym przypadku, wykonanie instrukcji, o których mowa w poprzednim podrozdziale. Następnie kopiujemy podane niżej pliki do odpowiednich lokalizacji.

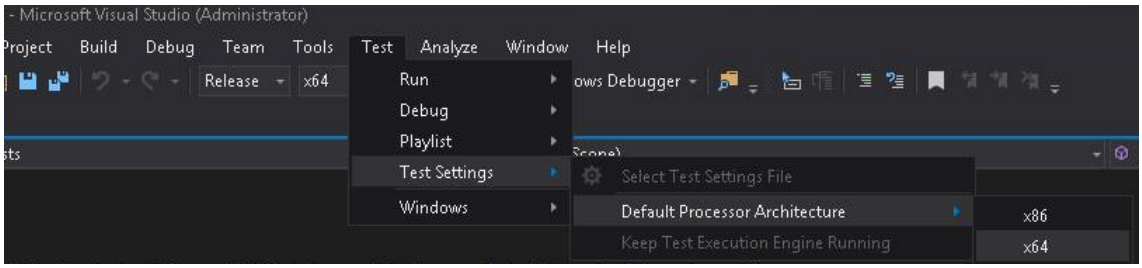
- Pliki "mpir.lib" i "mpir.dll" z "x64/Release" do "Microsoft Visual Studio 14.0/VC/lib"
- Plik "mpir.h" z "x64/Release" do "Microsoft Visual Studio 14.0/VC/include"
- Plik "vsyasm.exe" z plików instalacycjnych do "Microsoft Visual Studio 14.0/VC/bin"

W tym momencie możliwe jest już zbudowanie naszej biblioteki. Warto nadmienić, że jeżeli planujemy uruchomić testy jednostkowe, to musimy zmienić ich środowisko na 64-bitowe, w przeciwnym razie operacja ta zakończy się poniższym błędem.



Rysunek 3.1: Komunikat o błędzie w przypadku próby uruchomienia testów jednostkowych na niewłaściwej architekturze

By zmienić architekturę na 64-bitową, należy przestawić jej wartość domyślną dla środowiska testowego w sposób, w jaki zostało to przedstawione na rysunku niżej.



Rysunek 3.2: Ustawienie odpowiedniej architektury środowiska testowego

3.5 Instrukcja obsługi programu

Instrukcja dotyczy obsługi programu, który został poprawnie zainstalowany. Jeżeli nie zostało to wykonane, w celu jego pierwszego uruchomienia, niezbędne jest wcześniejsze wykonanie instrukcji przedstawionych w poprzednim podrozdziale.

Wszystkie komendy, na które pozwala nam aplikacja wykonujemy jako zapytania zdefiniowane w pojedynczej linii wejściowej. Zdefiniowałem dwa typy poleceń. Pierwszym z nich jest wprowadzenie wielomianu w wybranej przez siebie, poprawnej składniowo, postaci. Jedynym znaczącym ograniczeniem jest brak bezpośredniej możliwości wprowadzania typów zmienno przecinkowych. Jeżeli chcemy wprowadzić wartość ułamkową np. 1.4 musimy przedstawić jej wartość jako iloraz, czyli np. w postaci "14/10".

Drugim typem komendy jest zmiana domyślnych ustawień dla wybranych wartości liczbowych i dodatkowych funkcjonalności. Opcja ta pozwala użytkownikowi aplikacji na większą porcję informacji niż wartości pierwiastków rzeczywistych. By dana linia była traktowana jako wielomian wejściowy wystarczy po prostu wpisać jego wartość. Jeżeli zaś chcemy, by pozwoliła ona na zmiane wskazanej wartości musimy rozpocząć ją od wyrażenia "set". Dzięki temu sformułowaniu aplikacja będzie wiedziała, że dane zapytanie nie powinno być traktowane jako wartość wielomianu, ale przełączenie odpowiedniej opcji.

Poniżej zamieszczam listę wszystkich instrukcji, pozwalających na zmiane ustawień programu wraz z ich wartościami domyślnymi.

```
"set a {value}"
```

Wartość ta jest lewym krańcem przedziału, w którym aplikacja będzie znajdować pierwiastki rzeczywiste. Domyślną warotścią jest -1000. W przypadku próby ustawienia na wartość równą lub wyższą od aktualnej wartości b, polecenie takie zostanie zignorowane.

```
"seb b {value}"
```

Jest to wartość dla prawego krańca przedziału poszukiwań. Domyślną wartością jest 1000. Analogicznie jak dla wcześniejszego polecenia, komenda zostanie zignorowana jeżeli podana wartość będzie równa lub niższa od aktualnej wartości a.

```
"set precision {value}"
```

Jest to liczba cyfr po przecinku, które powinny zostać uwzględnione w zwracanym wyniku. Domyślną wartoscią jest wartość 6, oznaczająca zaokrąglanie pierwiastków wielomianu do wartości $(0.1)^6$.

```
"set type [{map}{vector}]" -> "set 1 [{m}{v}]"
```

Polecenie pozwala na wybranie typu wielomianu, który ma być wykorzystywany do obliczeń. Wartość "map" oznacza klasę PolynomialMap, zaś vector klasę PolynomialVector. Domyślnie wybierana jest mapa.

Warto zaznaczyć, że polecenie posiada też wersję skróconą. Ma to na celu łatwiejszą obsługę, poprzez brak konieczności wpiswania długich poleceń, w których bardzo łatwo jest popełnić błąd, który spowoduje, że dana linia wejściowa zostanie zignorowana. Dodatkowo spośród wartości znajdujących się wewnątrz nawiasów kwadratowych należy wybrać jedną i wpisać wnętrze danej pary nawiasów klamrarowych jako wartość w linii poleceń.

```
"set displaying roots [{1}{0}]" -> "set 2 [{1}{0}]"
```

Polecenie pozwala decydować, czy wartość obliczonych pierwiastków powinna być wyświetlana. Domyślnie ustawiona jest wartość 1, która powoduje, że każdy znaleziony pierwiastek rzeczywisty zostanie wypisany na ekran. Warto zauważyć, że pierwiastki wielokrotne zostaną wypisanę tylko jeden raz.

```
"set displaying sturm [{0}{1}]" -> "set 3 [{0}{1}]"
```

Komenda pozwala na wypisywanie ciagu Sturma, który powstał w trakcie znajdowania pierwiastków rzeczywistych badanego wielomanu. Domyślnie ustawiona jest wartość 0, czyli wielomiany ciągu Sturma nie są wypisywane.

```
"set displaying signs [{0}{1}]" -> "set 4 [{0}{1}]"
```

Polecenie pozwala na wypisywanie liczby zmian znaku w ciągu Sturma na obu krańcach badanego wielomianu. Wartością domyślną jest 0, co oznacza niewypisywanie tej informacji.

```
"set displaying after elimination [{0}{1}]" -> "set 5 [{0}{1}]"
```

Polecenie pozwala na wyświetlenie wielomianu po eliminacji jego pierwiastków wielokrotnych. Wartością domyślną jest 0, czyli brak wypisania tej informacji.

```
"set measuring time [{0}{1}]" -> "set 6 [{0}{1}]"
```

Komenda pozwala na zmierzenie czasu znajdowania pierwiastków wielomianu. Wypisywana wartość oznacza liczbę milisekund, którą trwało znalezienie wszystkich pierwiastków rzeczywistych. Domyślną wartością jest 0, oznaczające brak wykonywania pomiaru.

Jeżeli chcemy wyłączyć aplikację, program, poza standardową opcją wyłączenia, daje nam także dodatkową możliwość. Jeżeli dana linia będzie zawierała pojedynczy znak 'q' (od ang. "quit") program zakończy swoją pracę. Dodatkowo nastąpi to także w przypadku, gdy seria stu kolejnych zapytań będzie niepoprawna. Warto zaznaczyć, że wszystkie dokonane zmiany są trwałe tylko dla danego uruchomienia aplikacji. Po ponownym uruchomieniu ich ustawienia popnownie wracają do wartości domyślnych.

Rozdział 4

Przeprowadzone testy

4.1 Testy funkcjonalne

Przeprowadzone testy są połączeniem testów jednostkowych i funkcjonalnych. Zastosowałem nieco odmienne podejście od klasycznego, tzn. w moim przypadku najpierw powstał rdzeń aplikacji i dopiero wówczas zaczęły się pojawiać pierwsze testy. Spowodowane było to faktem, że dopiero wraz z rozwojem projektu zdałem sobie sprawę, że są one niezbędne do zapewnienia odpowiedniej jakości aplikacji, poprzez stałą weryfikację jej działania. Dodatkowo, regularna egzekucja istniejących już testów daje twórcy aplikacji informację o ewentualnej regresji i możliwość szybkiej korekty wprowadzonych zmian.

Testy funkcjonalne często zwane są testami czarnej skrzynki i wykonywane nie przez twórców aplikacji, a przez zewnętrznych testerów. W tym przypadku jest jednak inaczej i służą one wyłącznie programiście. Do tego celu nie zostało wykorzystane żadne zaawansowane środowisko testowe. Użyłem do tego framework, wbudowany w Microsoft Visual Studio 2015, służący do pisania testów jednostkowych. Okazał on się bardzo intuicyjny i prosty w obsłudze, a do mojego zastosowania całkowicie wystarczający. Do wszystkich testów wykorzystałem możliwość łatwego porównywania wartości typów string. Pozwoliło mi to na łatwe debugowanie testów i wyraźną informację, o tym jaka wartość była oczekiwana, a jaka została zwrócona przez wołaną funkcję. Ważnym aspektem było to, że wszystkie testowane klasy i funkcje aplikacji, w łatwy sposób, mogą być rzutowane na typ znakowy.

Testowane funkcje i powstałe klasy testowe zostały wybrane tak, by w jednoznaczny sposób móc zlokalizować przyczynę błędu. Wyniki testów zawsze powinny być czytane w ściśle określonej kolejności. Została ona ustalona tak, by funkcje niezależne od innych były wykonywane najpierw, a dopiero w przypadku ich poprawnego działania, następowała analiza tych bardziej złożonych. Tak więc, na początku testowane jest parsowanie łańcucha znaków do postaci wielomianów, następnie poszczególne operatory działań na wielomianach, by na końcu przeprowadzić weryfikację zasadniczej części programu, tzn. znajdowania pierwiastków.

Przeprowadzane testy programu weryfikują poprawne działanie obu struktur jednocześnie. In-

nymi słowy test daje wynik pozytywny wyłącznie wtedy, kiedy zwrócony wynik w przypadku obu typów PolynomialMap i PolynomialVector jest poprawny. Jest to parametr stosunkowo łatwy do zmienienia w kodzie programu, ale ustaliłem, że taka postać testów będzie najbardziej optymalna. W trakcie dewelopmentu aplikacji i debugowania, testy zostały zmieniane tak, by testować osobno daną funkcjonalność dla konkretnego typu wielomianu. Był to niezbędny krok, by komunikat zwracany przez framework testowy był jasny i czytelny. Dodatkowo warto zauważyć, że informacja o tym, że tylko jedna ze struktur działa, dostarcza wiele danych programiście. Dzięki temu, lista przyczyn usterki jest mocno ograniczona, co zazwyczaj bardzo przyśpiesza jej znalezienie i naprawienie.

4.1.1 ParserUniform

Klasa ta pozwala na weryfikację metody UniformInputString, mającej na celu unifikację otrzymanej wartości znakowej, przedstawiającej wielomian. Testy dla niej zostały napisane, pomimo że funkcja nie jest bardzo złożona. Jej funkcjonalność została uznana jednak za podstawową, a jej poprawne działanie jest niezbędne, by korzystać z programu. Jest to jedyna funkcja, która jest wołana niezależnie od innych.

Jej zadaniem jest nie tylko ujednolicić wprowadzaną postać wielomianu, ale przede wszystkim dokonać walidacji jej poprawności. Musi być ona odporna na różnego rodzaju błędy, wprowadzane przez użytkownika, zarówno przypadkiem, jak i celowo. Jej celem jest przeanalizować i zwrócić pusty obiekt typu string, w momencie, gdy nie uda się jej zrozumieć i w pełni przeanalizować otrzymanego wejścia. Funkcja ConvertToPolynomial wywołuje ją na początku, a następnie bazując na otrzymanym rezultacie, tworzy wybrany obiekt typu PolynomialMap lub PolynomialVector.

Testy tej funkcji zakładają sprawdzenie, że dla danego wejścia, zostaje zwrócone poprawne wyjście, czyli takie, które jest standardowe i łatwe do sparsowania. W przeciwnym wypadku, zwrócony ma zostać pusty łańcuch. Niezależnie jakie dane zostaną podane na wejściu, program nie ma prawa zakończyć się błędem, rzucając wyjątek, ani w ogóle się nie skończyć.

Na przedstawioną funkcjonalność napisałem ponad 20 testów funkcjonalnych. Poniżej zamieszczam przykładowe z nich, przedstawiając je w postaci – wejście, wyjście.

```
"" -> ""
"2x*+4" -> ""
"3*x-2" -> "3*x-2"
"4x" -> "4*x"
"x2+5" -> "x^2+5"
"10x3 - 4 x2 + 5x" -> "10*x^3-4*x^2+5*x"
"(3x-1)(2x+4)+2(x-4)" -> "(3x-1)*(2x+4)+2*(x-4)"
```

4.1.2 ParserConvert

Jest to klasa pozwalająca na testowanie metody ConvertToPolynomial. Wywołuje ona funkcję UniformInputString i analizuje jej wartość. Gdy jest to pusta wartość, tworzy wielomian zerowy. W przeciwnym wypadku, analizuje ona uzyskany rezultat i na jego podstawie tworzy wybrany obiekt typu PolynomialMap lub PolynomialVector. Rodzaj stworzonego wielomianu jest podawany jako parametr funkcji. W zależności od niego, także wszystkie pośrednie wielomiany, przybierają żądaną formę.

Przetestowanie tej funkcji powinno być możliwie bardzo dokładne i zawierać różne przypadki graniczne. Bez poprawnego działania tej funkcji, użytkownik w żaden sposób nie może przewidzieć jaką wartośc będzie miał dany wielomian. Ewentualny błąd w tej funkcji może powodować nieprzewidziane zachowanie pozostałych. W zachowaniu funkcji uwzględniono przypadki niedopuszczalnych wielomianów, które jednak przeszły weryfikację poprawności funkcji UniformInputData. Są nimi między innymi: ujemna lub niecałkowita potęga wielomianu, czy dzielenie przez wielomian zerowy. Wejścia takie nie zostaną odrzucone, ponieważ zawierają one wielomiany poprawne składniowo, a nielegalne wartości zostaną wykryte dopiero w momencie dokonania obliczeń, wykonanywanych jako pierwsze, np. z powodu nawiasu.

By przetestować funkcję, napisałem blisko 50 różnych testów. Poniżej przedstawiam wybrane z nich przypadki testowe.

```
"123" -> "123"
"100x" -> "100x"
"x^2" -> "x2"
"-5*x3" -> "-5x3"
"4*x+2" -> "4x+2"
"(3*x)^2" -> "9x2"
"(-2)10" -> "1024"
"x*2x" -> "2x2"
"(x+1)(x-1)" -> "x2-1"
"(x+1)(x+1)" -> "x2+2x+1"
"(x-1)^3" -> "x3-3x2+3x-1"
"(x3-1)*0" -> "0"
"(x-1)^3+4x2-x2+0*(x3-1)" -> "x3+3x-1"
"5^0" -> "1"
"(2x-3x+x)^1" -> "0"
"5x/2*3" -> "7.5x"
"(7x+5)/2" -> "7x/2+5/2"
```

4.1.3 Operatory dodawania, odejmowania i mnożenia

PolynomialSumOperator

Jest to klasa testowa odpowiedzialna za testowanie funkcjonalne operatora dodawania. W klasie Polynomial został przeciążony operator sumy, pozwalający dodawać dwa wielomiany, zapisując działanie to w czytelny i przejrzysty sposób. Testowanie poprawności tego działania polega na wywołaniu funkcji weryfikującej z trzema parametrami. Dwa pierwsze są składnikami dodawania, a ostatni to oczekiwana suma. Funkcja za pomocą operatora dodawania oblicza sumę wielomianów, a następnie porównuje ją z oczekiwanym wynikiem. Działanie polega na dodaniu wartości wyrazów, o tych samych potęgach, czyli redukcji wyrazów podobnych. Poniżej zamieszczam przykładowe testy.

```
"1" + "20" = "21"
"5" + "-11" = "-6"
"12x" + "0" = "12x"
"12x3" + 15x3" = "27x3"
"12x3+4x2" + "-15x2" = "12x3-11x2"
"12x3+4x+8" + "-15x2+30x+5x-5-3" = "12x3-15x2+39x"
```

PolynomialSubOperator

Jest to bliźniacza klasa testowa dla PolynomialSumOperator. Jedyną zmianą jest to, że w tym przypadku, podajemy kolejno: odjemną, odjemnik i oczekiwaną różnicę. Warto zauważyć, że odejmowanie dwóch wielomianów, podobnie jak w przypadku liczb, sprowadza się do zsumowania pierwszego z nich i wartości przeciwnej do drugiego. Przeprowadzone testy są analogiczne, jak dla operatora sumy.

```
"1" - "20" = "-19"
"5" - "-11" = "16"
"12x" - "0" = "12x"
"12x3" - 15x3" = "-3x3"
"12x3+4x2" - "-15x2" = "12x3+19x2"
"12x3+4x+8" - "-15x2+30x+5x-5-3" = "12x3+15x2-31x+16"
```

PolynomialMulOperator

Jest klasą, której zadaniem jest, jak najdokładniej przetestować operator mnożenia wielomianów. Testy analogicznie jak w przypadku poprzednich klas, mają na celu sprawdzenie, czy dla dwóch podanych czynników, uzyskamy pożądany wynik. Operacja iloczynu wielomianów polega na przemnożeniu wszystkich wyrazów pierwszego z nich przez wyrazy drugiego. Mnożąc dwa wyrazy, mnożymy przez siebie ich wartości, zaś wykładniki sumujemy. Następnie dokonujemy redukcji wyrazów podobnych, poprzez ich dodanie, a uzyskany w ten sposób wielomian jest szukanym iloczynem.

```
"1" * "20" = "20"
"0" * "-55" = "-55"
"-2x" * "7" = "-14x"
"12x" * "-5x" = "-60x2"
"12x3" * "15x3" = "180x6"
"12x3+4x2" * "-15x2" = "-180x5-60x4"
"12x3+4x+8" * "-15x2+35x-8" = "-180x5+420x4-156x3+20x2+248x-64"
```

4.1.4 Operatory dzielenia i modulo

PolynomialDivOperator

PolynomialDivOperator jest klasą testową, pozwalającą na przetestowanie poprawności dzielenia wielomianów. Funkcja ją weryfikująca przyjmuje trzy parametry. Są nimi dzielna, dzielnik oraz spodziewany iloraz. Podany dzielnik nie może być wielomianem zerowym, ponieważ dzielenie przez 0, również w przypadku wielomianów, nie jest dopuszczalne. Operacja dzielenia jest, w przypadku wielomianów, najbardziej skomplikowaną operacją. Algorytm jej wykonania przypomina nieco dzielenie pisemne w przypadku liczb całkowitych. Wyraz stojący przy najwyższej potędze dzielnej, jest dzielony przez wyraz stojący przy najwyższej potędze dzielnika, a otrzymany wynik jest zapisywany. Następnie otrzymany wyraz jest mnożony przez wielomian przeciwny do dzielnika, a otrzymany rezultat sumowany z dotychczasową wartością wielomianu. Uzyskany wynik staje się nową dzielną, a sytuacja powtarza się do momentu, gdy stopień dzielnej będzie niższy, niż stopień wielomianu dzielącego. Wartości otrzymane z kolejnych dzieleń tworzą wyrazy ilorazu, o którym wiemy, że jego stopień jest równy różnicy stopni argumentów operatora dzielenia.

```
"20" / "20x" = "0"
"20" / "-1" = "=-20"
"-21x" / "7x" = "-3"
"-12x2" / "-6x" = "2x"
"x3+x2+x" / "x" = "x2+x+1"
"-30x3-15x" / "-5x2" = "6x"
"x6-6x4-4x3+9x2+12x+4" / "x5-4x3-2x2+3x+2" = "x"
"3x2+4" / "2x" = "3x/2"
```

PolynomialModOperator

Jest klasą testową dla operatora modulo, którego obliczanie oparte jest na wykonywaniu operacji dzielenia, z tymże w tej sytuacji zwracany jest inny wielomian. W przypadku opisanym powyżej, tzn. w momencie, gdy stopień dzielnej jest większy niż stopień aktualnie przetwarzanego wielomianu, to drugi z tych wielomianów nazywany jest resztą z dzielenia. Wiemy o nim, że jego stopień jest mniejszy niż dzielnika. W szczególnym przypadku, gdy dzielnik jest podzielny przez dzielną, uzyskana reszta jest wielomianem zerowym. Warto zaznaczyć, że podobnie jak w przypadku

dzielenia, w celu weryfikacji podawane są także trzy wielomiany. Ostatni z nich jest pożądanym wynikiem operacji modulo, zaś środkowy dzielnikiem, co wymusza, by jego wartość była różna od zera. Poniżej zaprezentowane zostały te same przypadki testowe jak dla dzielenia, ale w tej sytuacji wynikiem jest rezultat operacji modulo. Dodatkowo lista testów została powiększona o testy, charakteryzujące się niezerową resztą.

```
"20" % "20x" = "20"
"20" % "-1" = "=0"
"-21x" % "7x" = "0"
"-12x2" % "-6x" = "0"
"x3+x2+x" % "x" = "0"
"-30x3-15x" % "-5x2" = "-15x"
"x6-6x4-4x3+9x2+12x+4" % "x5-4x3-2x2+3x+2" =
 = "-2(x4+x3-3x2-5x-2)"
"x3+x2+x1+11" % "x2+1" = "10"
"(x+2)^3+1" % "x+2" = "1"
"x4-4x3+6x2-7x+3" % "(x-1)^2" = "-3x+2"
"3x2+4" % "2x" = "4"
```

4.1.5 PolynomialDerivative

Zadaniem tej klasy jest sprawdzenie, czy metoda pozwalająca na obliczanie pochodnej wielomianu działa poprawnie. Jest ona wykonywana na konkretnej instancji obiektu typu Polynomial i zwraca referencję do jego pochodnej. Funkcja bazuje na operatorach dodawania i mnożenia. Ich poprawne działanie jest zatem niezbędne, by metoda ta mogła poprawnie funkcjonować. Weryfikacja poprawności wyników następuje poprzez porównanie oczekiwanej wartości pochodnej wielomianu z wartością obliczoną. Poniżej prezentuję wybrane przypadki testowe dla tej funkcjonalności.

```
("5")' = "0"
("2x")' = "2"
("x2")' = "2x"
("7x3")' = "21x2"
("120x130")' = "15600x129"
("-x101+2x17+x4+x")' = "-101x100+34x16+4x3+1"
```

4.1.6 PolynomialAfterElimination

Jest to klasa weryfikująca, czy wartość wielomianu, po eliminacji pierwiastków wielokrotnych, jest poprawna. Metoda ta jest wykonywana dla danego obiektu typu Polynomial i zwraca referencję do wielomianu wynikowego. Jej wartość nie jest jednak bezpośrednio porównywana z oczekiwanym wynikiem. Wcześniej jest bowiem wykonywana jeszcze funkcja normalizująca uzyskany wielomian. Jest to niezbędne, gdyż w przeciwnym razie, wykonywane porównanie mogłoby wskazać fałszywy

rezultat. Jest to spowodowane tym, że funkcja porównująca, ze względów wydajnościowych, nie normalizuje danych wielomianów. Decyzja taka była umotywowana faktem, że w takim przypadku porównanie całkowicie odmiennych wielomianów byłoby liniowe ze względu na ich stopień, zamiast odbywać się w czasie stałym. Aplikacja zakłada, że w każdym miejscu, gdzie jest to wymagane, będzie następowało jawne wywołanie odpowiedniej funkcji.

Aspektem bardzo ułatwiającym testowanie tej funkcjonalności jest możliwość podawania wielomianu wejściowego w postaci iloczynu wielu czynników. Pozwala to podawać kolejne czynniki, zawierające dane pierwiastki wielomianu, z określonymi krotnościami i spodziewać się dokładnie tych samych pierwiastków, ale jednokrotnych. Dzięki temu jest się niezależnym od innych programów, pozwalających na obliczanie zer wielomianów, ponieważ spodziewane wartości, zawarte są pośrednio w wielomianie wejściowym. Poniżej zamieszczam przykładowe testy, z uwzględnieniem podawania wejścia, w różnej postaci.

```
"20" -> "1"
"2x" -> "x"
"(x-1)^2" -> "x-1"
"(x+3)^4" -> "x+3"
"(x+1)(x+2)" -> "(x+1)(x+2)"
"x+3)^2*(x+1)" -> "(x+3)(x+1)"
"x3*(x+2)^10" -> "x(x+2)"
"(x-1)^2*(x+1)^2*(x-2)^2" -> "(x-1)(x+1)(x-2)"
"(x+1)^4*(x2+x+1)" -> "(x+1)(x2+x+1)"
"(2x+3)^4*(x2+2x+1)" -> "(2x+3)(x+1)/2")
```

4.1.7 PolynomialValue

Jest klasą weryfikującą, czy wartość wielomianu w danym punkcie jest obliczana poprawnie. Funkcja weryfikująca przyjmuje dwa argumenty wejściowe – wielomian oraz dany punkt. Otrzymany wynik porównuje z trzecim parametrem, którym jest oczekiwany rezultat. Poza sprawdzeniem poprawności działania samego wielomianu, weryfikowane jest także działanie klasy Number. Ważnym aspektem jest tutaj, przede wszystkim, sprawdzenie zachowania funkcji, w przypadku dużych liczb, które często pojawiają się w momencie obliczania wartości wielomianów wysokich stopni. Należy wziąć pod uwagę, że obliczenie wartości w punkcie $x_0 = 10$, dla wielomianu stopnia setnego wymaga liczby posiadającej aż 101 cyfr. To właśnie z tego względu w projekcie niezbędna była biblioteka mpir, pozwalająca na posługiwanie się takimi wartościami liczbowymi. Poniżej prezentuję przykładowe testy na obliczanie wartości wielomianu w danym punkcie.

```
W(x) = "20", x0 = 0 -> W(0) = 20
W(x) = "x", x0 = 13 -> W(13) = 13
W(x) = "12x2", x0 = 3 -> W(3) = 108
W(x) = "(x+1)(x+2)", x0 = 3 -> W(3) = 20
W(x) = "(x+3)^2*(x+1)", x0 = -2 -> W(-2) = -1
```

```
W(x) = "(x+1)^4*(x-2)^2", x0 = 2 -> W(2) = 0
W(x) = "(x-1)^2*(x+1)^4*(x-2)^2", x0 = 3 -> W(3) = 1024
W(x) = "(x+1)^4*(x2+x+1)", x0 = 2 -> W(2) = 567
W(x) = "x39+16x+x8", x0 = 2 -> W(0) = 0x8000010100
W(x) = "(4x+2)^2*(6x+1)", x0 = 1.5 -> W(1.5) = 640
```

4.1.8 PolynomialNumberOfRoots

Jest to klasa, sprawdzająca ile pierwiastków rzeczywistych znajduje się w podanym przedziale. Pod uwagę brane są wyłącznie różne pierwiastki, dlatego na początku funkcji weryfikującej wywoływana jest funkcja PolynomialAfterEliminationOfMultipleRoots, a dopiero na jej znormalizowanym rezultacie wykonywana jest właściwa funkcja. Jej rezultat porównywany jest ze spodziewaną wartością, podawaną w postaci zmiennej typu int. Domyślnymi granicami przedziału są wartości niewłaściwe $-\infty$ i $+\infty$, więc gdy interesują nas pierwiastki w całym zbiorze liczb rzeczywistych, nie ma potrzeby ich podawać. W przeciwnym razie są one nadpisywane wartościami pól klasy – a i b, oznaczających lewy i prawy kraniec przedziału. Możliwe jest także sprecyzowanie granicy przedziału tylko z jednej strony, np. gdy szukamy liczby pierwiastków dodatnich. Poniżej zamieszczam przykładowe testy dla różnych przedziałów wyszukiwania.

```
W(x) = "20" -> 0
W(x) = "2x" -> 1
W(x) = "-x3" -> 1
W(x) = "(x-1)^2" -> 1
W(x) = "(x+1)(x+2)" -> 2
W(x) = "(x+3)^2*(x+1)^3" -> 2
W(x) = "(x-1)^2*(x+1)^2*(x-2)^2" -> 3
W(x) = "(x+1)^4*(x2+x+1)" -> 1
W(x) = "12x2", a = -1 -> 1
W(x) = "x3*(x+2)^10", a = -1, b = 4 -> 1
W(x) = "(x+1)^4*(x-2)^2", a = 0, b = 1 -> 0
W(x) = "(2x-3)^2*(2x+1)^2*(2x-7)^2", a = 2 -> 1
```

4.1.9 PolynomialRoots

Na początku funkcji weryfikującej dokonywana jest eliminacja pierwiastków wielokrotnych. Otrzymany wielomian poddawany jest normalizacji, a wówczas znajdowane są jego pierwiastki w podanym przedziale. Podobnie jak w przypadku klasy PolynomialNumberOfRoots, domyślnym przedziałem jest zakres od $-\infty$ do $+\infty$. Pierwiastki w wektorze wyjściowym funkcji znajdują się w kolejności ich znalezienia. Także w tym przypadku, konieczne jest więc ich posortowanie.

Poniżej zamieszczam większość testów, sprawdzających ostateczną funkcjonalność programu. Lista ta jest bardziej rozbudowana niż poprzednie, ponieważ uznałem, że jest ona zdecydowanie najistotniejszą częścią testów funkcjonalnych mojego programu.

```
W(x) = "20" -> {}
W(x) = "2x" -> {0}
W(x) = "-x3" -> {0}
W(x) = "(x-1)^2" -> {0}
W(x) = "(x+1)(x+2)" -> {-2, -1}
W(x) = "(x+3)^2*(x+1)^3" -> {-3, -1}
W(x) = "(x-1)^2*(x+1)^2*(x-2)^2" -> {-1, 1, 2}
W(x) = "(x+1)^4*(x2+x+1)" -> {-1}
W(x) = "12x2", a = -1 -> {0}
W(x) = "x3*(x+2)^10", a = -1, b = 4 -> {0}
W(x) = "(x+1)^4*(x-2)^2", a = 0, b = 1 -> {}
W(x) = "(x-1)*(5x-7)^2*(2x-3)", a = -10, b = 1- -> {1, 7/5, 5/3}
```

4.2 Testy wydajnościowe

Celem niniejszego podrozdziału jest przeprowadzenie testów wydajnościowych dla obu typów wielomianów. PolynomialMap i PolynomialVector to bardzo podobne klasy, których struktura została jednak oparta na odmiennych podejściach. PolynomialMap opiera się na przechowywaniu wyłącznie niezerowych współczynników wielomianu. Z kolei PolynomialVector jako kontenera danych używa wektora, co wymusza przechowywanie wszystkich współczynników, także zerowych.

Zestaw przeprowadzonych testów dobrałem tak, by pozwoliły one na zbadanie zupełnie różnych przypadków. Teoretycznie bowiem obiekty typu PolynomialMap powinny być dużo wydajniejsze dla wielomianów rzadkich. Z drugiej strony, w przypadku wielomianów gęstych różnica pomiędzy obiema klasami powinna być zdecydowanie mniejsza. Wydaje się, że w takiej sytuacji niewielką przewagę powinna mieć klasa wielomianu oparta o użycie tablicy. W tym miejscu warto przypomnieć, że wielomiany nazywami gęstymi, jeżeli większość współczynników jest niezerowa, zaś wielomiany, których większość współczynników stanowią zera to wielomiany rzadkie.

Zapoznajmy się więc z poniższymi testami, by zweryfikowac przedstawioną wyżej tezę.

4.2.1 Testowanie wielomianów gęstych

Wielomiany stopnia parzystego

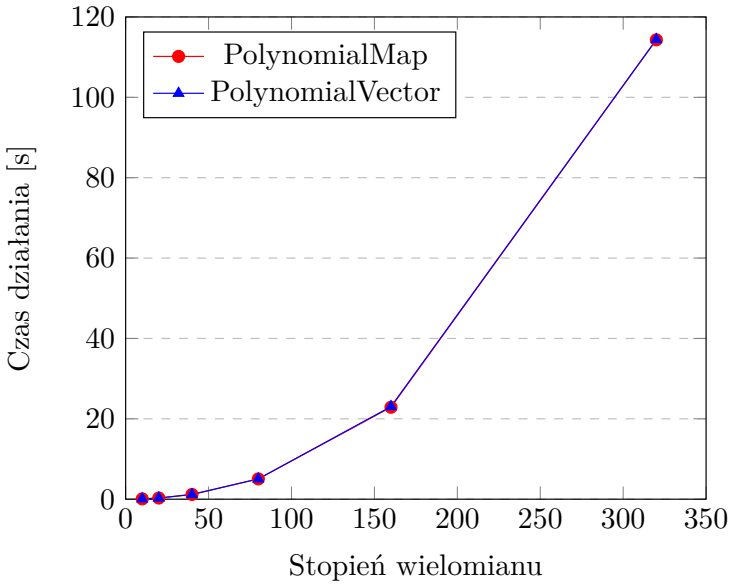
Pierwszym przypadkiem testowym będzie sprawdzenie czasów znajdowania pierwiastków dla takich wielomianów, których wszystkie współczynniki są równe 1. Testowane wielomiany można przedstawić wzorem:
$W(x) = x^k + x^{k-1} + x^{k-2} + ... + x^2 + x + 1$, gdzie k jest stopniem wielomianu W.

W przypadku, gdy stopień wielomianu jest liczbą parzystą, a jego wszystkie współczynniki są równe 1, wielomian nie posiada pierwiastków rzeczywistych. Sprawdźmy ile czasu zajmie każdej ze struktur stwierdzenie braku pierwiastków rzeczywistych dla wielomianów powyższej postaci.

Testowany wielomian	Czas dla PolynomialMap [s]	Czas dla PolynomialVector [s]	Współczynnik czasów
$W(x) = x^{10}+x^9+...+x+1$	0.077	0.079	1.026
$W(x) = x^{20} + x^{19} + ... + x + 1$	0.29	0.291	1.003
$W(x) = x^{40} + x^{39} + ... + x + 1$	1.176	1.174	0.998
$W(x) = x^{80} + x^{79} + ... + x + 1$	5.046	5.048	1
$W(x) = x^{160} + x^{159} + ... + x + 1$	22.879	23.047	1.007
$W(x) = x^{320} + x^{319} + ... + x + 1$	114.311	114.382	1.001

Tabela 4.1: Porównanie czasów znajdowania pierwiastków dla gęstych wielomianów parzystego stopnia

W tabeli dla każdego przeprowadzonego testu została dodana wartość współczynnika czasów, zdefiniowa jako iloraz czasu działania typu PolynomialVector i PolynomialMap. Ich wartość dla wszystkich testów jest bliska lub równa 1. Na tej podstawie łatwo zauważyć, że czasy znajdowania pierwiastków wielomianów dla obu struktrur jest praktycznie identyczny. Z tego powodu poniższy wykres pokazuje dwie praktycznie w całości pokrywające się linie, oznaczające czas działania obu typoów wielomianów w zależności od testowanego stopnia.



Rysunek 4.1: Czasy znajdowania pierwiastków dla gęstych wielomianów parzystego stopnia

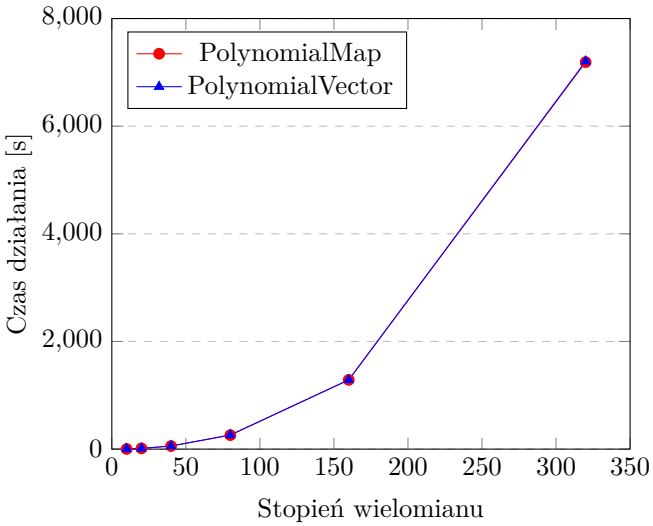
Wielomiany stopnia nieparzystego

To analogiczny test w porównaniu z poprzednim, jednak w tym przypadku stopień wielomianu jest nieparzysty. Wpływa to na liczbę pierwiastków rzeczywistych wielomianu, zwiększając ją o 1. Sprawdźmy, czy wyniki będą podobne do tych uzyskanych dla wielomianów stopnia parzystego.

Testowany wielomian	Czas dla PolynomialMap [s]	Czas dla PolynomialVector [s]	Współczynnik czasów
$W(x) = x^{11} + x^{10} + ... + x + 1$	3.224	3.269	1.014
$W(x) = x^{21} + x^{20} + ... + x + 1$	13.384	13.548	1.012
$W(x) = x^{41} + x^{40} + ... + x + 1$	57.336	57.336	1
$W(x) = x^{81} + x^{80} + ... + x + 1$	260.175	261.915	1.007
$W(x) = x^{161} + x^{160} + ... + x + 1$	1286.409	1286.261	1
$W(x) = x^{321} + x^{320} + ... + x + 1$	7190.234	7205.386	1.002

Tabela 4.2: Porównanie czasów znajdowania pierwiastków dla gęstych wielomianów nieparzystego stopnia

Także i w tym przypadku porównywane czasy działania są prawie identyczne. Ich wartości w zależności od stopnia testowanego wielomianu zostały przedstawione na poniższym wykresie.



Rysunek 4.2: Czas znajdowania pierwiastków dla wielomianów zawierających kolejne pierwiastki całkowite

Porównując wartości uzyskane dla wielomianów stopnia parzystego i nieparzystego uwagę zwraca fakt, że uzyskane czasy dla wielomianów stopnia nieparzystego, czyli takich posiadających przynajmniej jeden pierwiastek rzeczywistych są kilkadziesiąt razy większe niż dla wielomianów stopnia parzystego. Oznacza to, że liczba pierwiastków rzeczywistych ma ogromny wpływ na czas działania algorytmu ich znajdowania. Testy potwierdzają, że sytuacja taka jest obserwowana niezależnie od typu użytej struktury wielomianu.

4.2.2 Testowanie wielomianów rzadkich

Wielomiany stopnia parzystego

Kolejnym przypadkiem jest zbadanie rzadkich wielomianów, nieposiadających pierwiastków całkowitych. By zapewnić ten element, najłatwiejszym sposobem jest konstrukcja wielomianu postaci $ax^{2k} + b$, gdzie $a \in R_+, b \in R, k \in Z_+$. Dzięki temu, mamy pewność, że wielomian nie posiada pierwiastków całkowitych, a dodatkowo ma najmniejszą możliwą liczbę niezerowych współczynników.

Teoretycznie, wielomiany rzadkie powinny się charakteryzować największą dysproporcją w wydajności pomiędzy typami PolynomialMap i PolynomialVector, na korzyść tego pierwszego. Jest to spowodowane faktem, że to właśnie w tym przypadku, odsetek zerowych współczynników jest największy. Czynnik ten powinien bardzo negatywnie wpływać na czas znajdowania pierwiastków, a w tym przypadku na stwierdzenie, o ich braku. Zapoznajmy się z wynikami testów, by zweryfikować postawioną powyżej tezę.

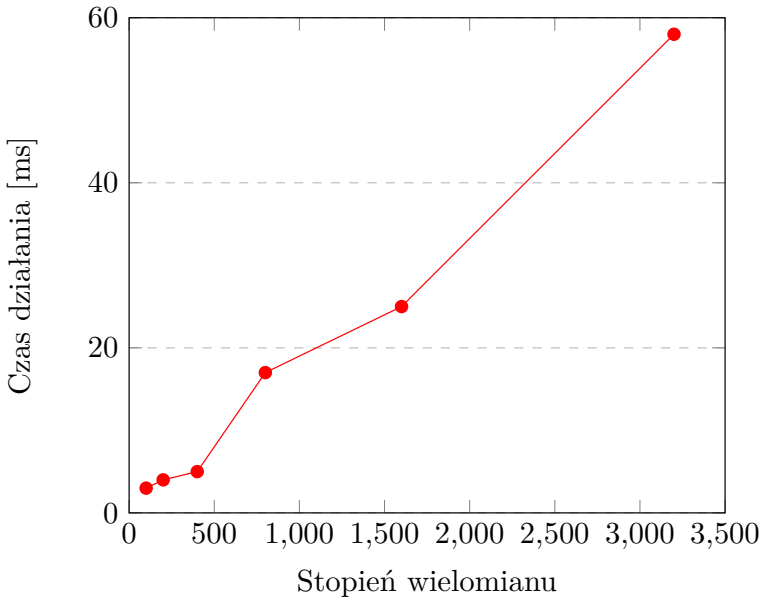
Testowany wielomian	Czas dla PolynomialMap [s]	Czas dla PolynomialVector [s]	Współczynnik czasów
$W(x) = x^{100} + 1$	0.003	0.045	15
$W(x) = x^{200} + 1$	0.004	0.159	39.75
$W(x) = x^{400} + 1$	0.005	0.647	129.4
$W(x) = x^{800} + 1$	0.017	2.790	164.12
$W(x) = x^{1600} + 1$	0.025	13.035	521.4
$W(x) = x^{3200} + 1$	0.058	67.311	1160.53

Tabela 4.3: Porównanie czasów znajdowania pierwiastków dla rzadkich wielomianów parzystego stopnia

Jak widać na podstawie uzyskanych czasów dla obu wielomianów dysproporcja pomiędzy ich wydajnością dla wielomianów rzadkich jest ogromna. Można zauważyć, że wraz ze wzrostem wielomianów staje się ona coraz większa. Dodatkowo, widać, że w przypadku PolynomialVector uzyskane czasy rosną około pięciokrotnie, przy dwukrotnym wzroście stopnia wielomianu, co oznacza przekraczającą nawet kwadratową zależność.

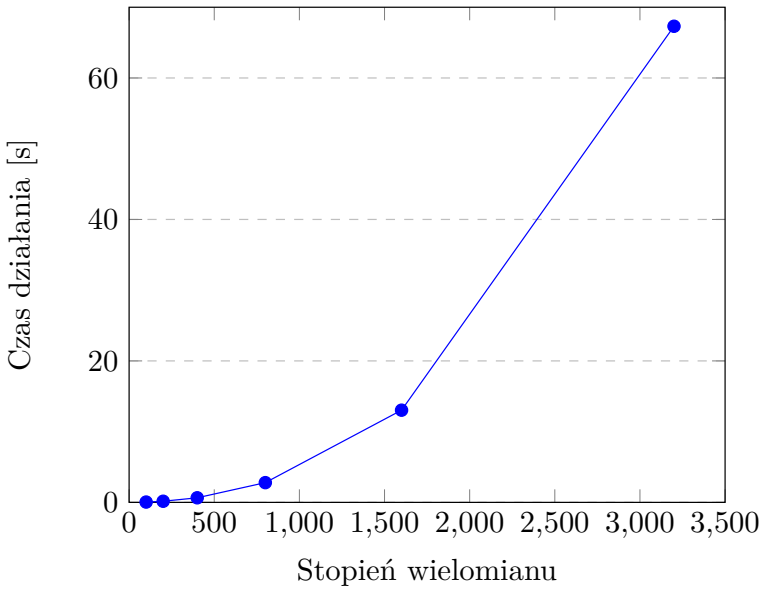
Poniżej przedstawiam wykresy dla obu typów wielomianów. Zdecydowałem się na rozdzielenie ich na dwa osobne wykresy ze względu na wspomnianą wcześniej ogromną dysproporcję. W przeciwnym razie na wykresie o liniowej skali wartości czasów dla PolynomialMap byłyby tak małe, że

pokrywałyby się z osią poziomą.



Rysunek 4.3: Czasy znajdowania pierwiastków przez PolynomialMap dla rzadkich wielomianów parzystego stopnia

Warto zauważyć, że wszystkie czasy przedstawione na powyższym wykresie zostały wyrażone w milisekundach. Dodatkowo na uwagę zasługuje fakt, że również czas działania dla wielomianów bardzo wysokich stopni jest bardzo mały. Wartości te rosną w tempie nieco wyższym niż liniowe. Oznacza to bardzo dobrą skalowalność działania algorytmu dla struktury PolynomialMap.



Rysunek 4.4: Czasy znajdowania pierwiastków przez PolynomialVector dla rzadkich wielomianów parzystego stopnia

Wielomiany stopnia nieparzystego

Teraz rozpatrzmy sytuację, w której ponownie mamy do czynienia z wielomianem rzadkim, o którym wiemy, że posiada przynajmniej jeden pierwiastkek rzeczywisty. By to zapewnić, wystarczy skonstruować wielomian w postaci $ax^{2k+1}+b$, gdzie $a \in R \setminus \{0\}, b \in R, k \in Z_+$. Jak wiemy na podstawie przedstawionego w drugim rozdziale twierdzenia, każdy wielomian stopnia nieparzystego posiada przynajmniej jeden pierwiastek rzeczywisty.

Wydaje się, że podobnie jak w przypadku wielomianów nieposiadających pierwiastków, również w tym, czas znajdowania pierwiastków dla typu PolynomialMap powinien być znacznie krótszy niż dla PolynomialVector. Natomiast, czy aby na pewno tak jest? Rozważmy poniższy przykład.

Postarajmy się zbudować ciąg Sturma dla wielomianu $x^{101}+1$. Zacznijmy od obliczenia pochodnej. $(x^{101}+1)' = 101x^{100}$. Obliczona wartość jest drugim elementem ciągu Sturma. Obliczmy resztę z dzielenia dwóch pierwszych wielomianów tego ciągu:

$$\begin{array}{l} \frac{1}{101}x \\ \hline (x^{101}+1) : (101x^{100}) \\ \underline{-\,x^{101}} \\ \hline 1 \end{array}$$

Trzecim i jednocześnie ostatnim wyrazem ciągu Sturma jest więc -1, czyli wartość przeciwna do otrzymanej reszty. Dysponujemy zatem trzyelementowym ciągiem Sturma o elementach $x^{101}+1, 101x^{100}, -1$. Obliczmy teraz liczbę zmian znaków dla bardzo dużych wartości bezwzględnych, dążących odpowiednio do $-\infty$ i $+\infty$. W prawym krańcu przedziału będą to wartości $+, +, -$, zaś w lewym $-, +, -$. Otrzymaliśmy jedną zmianę znaku dla wartości bliskiej $-\infty$ i dwie zmiany dla wartości bliskiej $+\infty$. Oznacza to, że liczba pierwiastków rzeczywistych jest równa różnicy tych zmian w lewym i prawym krańcu, czyli w tym przypadku wynosi 1.

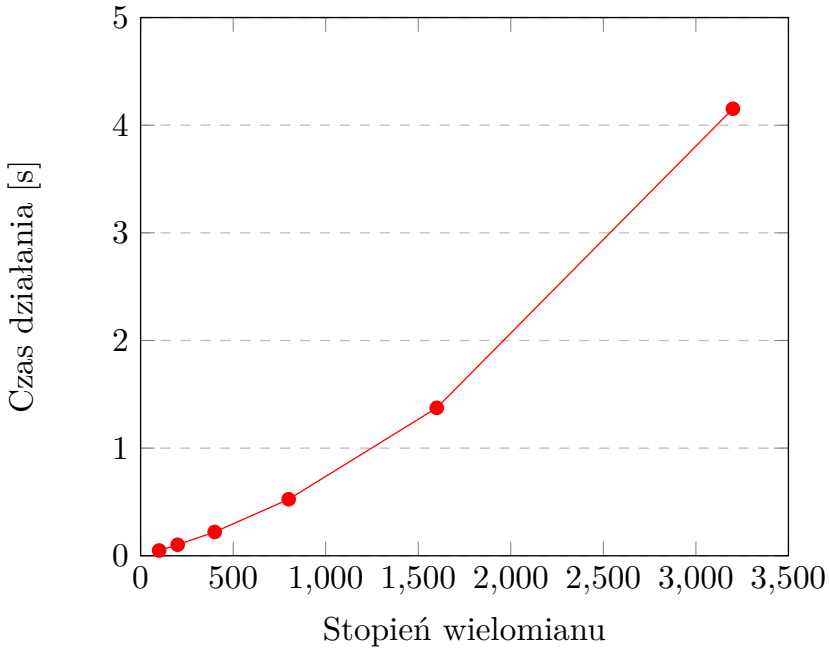
Jak widać, dwa z trzech elementów ciągu Sturma to rzadkie wielomiany wysokich stopni. Na takiej podstawie, wydaje się, że powyższa teza została potwierdzona. Jednocześnie jednak, warto zauważyć, że dla tego rodzaju wielomianów liczba koniecznych operacji jest niewielka, z uwagi na znikomą liczbę elementów ciągu Sturma. Wydaje się zatem, że to właśnie ich liczba bezpośrednio wpływa na czas znajdowania pierwiastków, a ich charakterystyka, czyli liczba niezerowych współczynników ma największe znaczenie dla wydajności obu typów wielomianów. Sprawdźmy, czy wyniki testów potwierdzają powyższą teorię.

Testowany wielomian	Czas dla PolynomialMap [s]	Czas dla PolynomialVector [s]	Współczynnik czasów
$W(x) = x^{101} + 1$	0.049	2.278	46.49
$W(x) = x^{201} + 1$	0.102	9.438	92.53
$W(x) = x^{401} + 1$	0.221	40.432	182.95
$W(x) = x^{801} + 1$	0.525	184.608	351.63
$W(x) = x^{1601} + 1$	1.374	912.716	664.28
$W(x) = x^{3201} + 1$	4.153	5125.106	1234.07

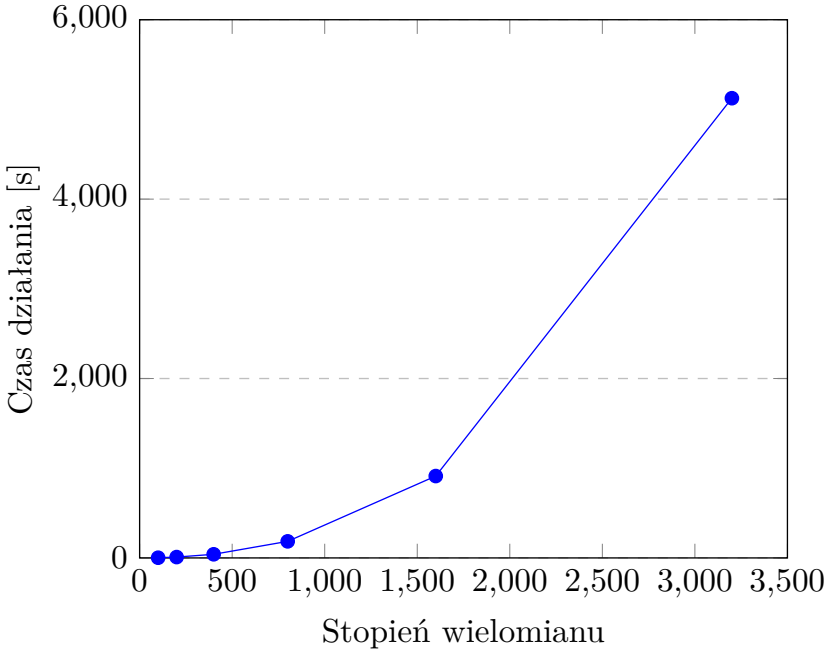
Tabela 4.4: Porównanie czasów znajdowania pierwiastków dla rzadkich wielomianów nieparzystego stopnia

Jak widać również w tym przypadku zauważalna jest niesamowita różnicą między wydajnością działania obu struktur. Dodatkowo również ten test potwierdza, że nieparzysty stopień wielomianu powoduje wielokrotne wydłużenie czasu trwania algorytmu.

Poniżej przedstawiam wykresy z czasami działania obu struktur.



Rysunek 4.5: Czasy znajdowania pierwiastków przez PolynomialMap dla rzadkich wielomianów nieparzystego stopnia



Rysunek 4.6: Czasy znajdowania pierwiastków przez PolynomialVector dla rzadkich wielomianów nieparzystego stopnia

Można zauważyć, że czasy działania w przypadku obu struktur są zdecydowanie krótsze niż dla wielomianów gęstych. Mniejszą różnicę obserwujemy dla typu PolynomialVector. Jej istnienie jest spowodowane faktem, że w obu przypadkach musimy przejrzeć kolejne współczynniki, ale większość reakcji wymaga dodatkowego działania wyłącznie wtedy, gdy są one niezerowe.

Zdecydowanie większa dysproporcja jest w przypadku klasy PolynomialMap. Wynika ona z konieczności dodatkowych obliczeń dla kolejnych współczynników, a oprócz tego wymusza przetrzymywanie informacji o ich wartości.

4.2.3 Badanie wielomianów posiadających wyłącznie pierwiastki rzeczywiste

Wielomiany o pierwiastkach całkowitych

Na początku zastanówmy się, jak sprawić, by liczba elementów ciągu Sturma była jak największa. Pozwoli to nam, na zmaksymalizowanie liczby operacji na wielomianach danego stopnia. Aby uzyskać taki wielomian, najłatwiej jest zapewnić, by miał on liczbę pierwiastków rzeczywistych równą lub zbliżoną do jego stopnia. Jak wiemy, liczba ta jest równa różnicy liczby zmian ciągu Sturma w obu przedziałach, a jej wartość jest ograniczona przez liczbę wyrazów tego ciagu. Wynika z tego, że liczba elementów ciagu Sturma jest zawsze większa od liczby pierwiastków rzeczywistych.

Teraz zastanówmy się, jak skonstruować wielomian, który zawiera liczbę różnych pierwiastków rzeczywistych równą jego stopniowi. Intuicyjne utworzenie wielomianu, podając jego kolejne

współczynniki jest niemożliwe dla wysokich stopni.

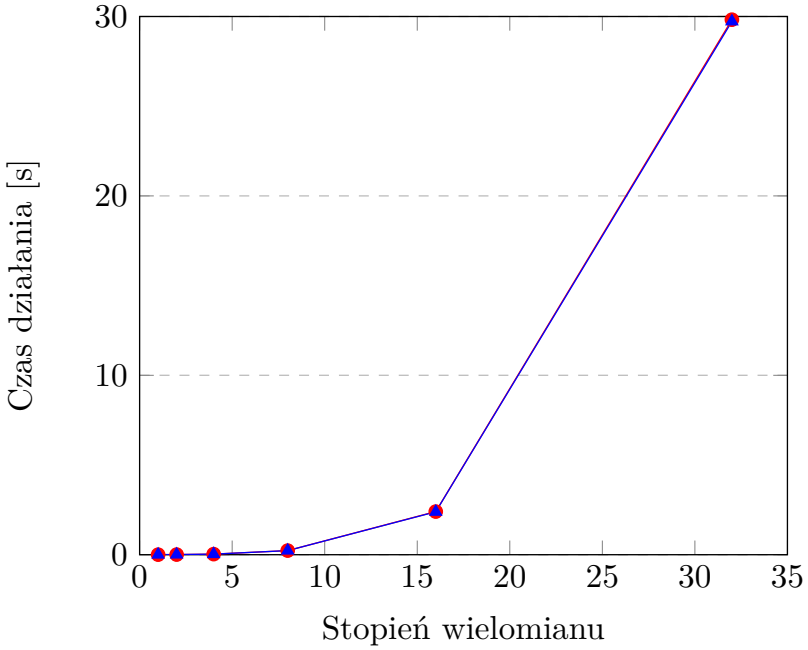
Rozwiązaniem jest więc przedstawienie go w postaci czynników, wskazując jego kolejne pierwiastki. Przykładowo wielomianowi W, o stopniu k, możemy przypisać pierwiastki, będące kolejnymi dodatnimi liczbami całkowitymi, ustalając jego wartość jako:
$W(x) = (x-1)*(x-2)*(x-3)*...*(x-(k-2))*(x-(k-1))*(x-k)$, gdzie k jest stopniem wielomianu.

Przetestujemy więc wielomiany powyższej postaci. Z powodu ograniczeń czasowych zdecydowałem się w tym teście na badanie wielomianów niższych stopni, niż w pozostałych przypadkach. Zapoznajmy się z rezultatami opisanego eksperymentu.

Testowany wielomian	Czas dla PolynomialMap [s]	Czas dla PolynomialVector [s]	Współczynnik czasów
$W(x) = x - 1$	0.003	0.003	1
$W(x) = (x-1)(x-2)$	0.007	0.008	1.143
$W(x) = (x-1)(x-2)(x-3)(x-4)$	0.032	0.032	1
$W(x) = (x-1)(x-2)*...*(x-7)(x-8)$	0.229	0.228	0.996
$W(x) = (x-1)(x-2)*...*(x-15)(x-16)$	2.404	2.391	0.995
$W(x) = (x-1)(x-2)*...*(x-31)(x-32)$	29.822	29.725	0.997

Tabela 4.5: Porównanie czasów znajdowania pierwiastków dla wielomianów zawierających kolejne pierwiastki całkowite

Widzimy, że podobnie jak w przypadku wielomianów gęstych zmierzone czasy działania są niemal identyczne dla obu struktur. Jednocześnie trzeba zauważyć, że duża liczba pierwiastków rzeczywistych wpływa na liczbę elementów ciągu Sturma. Powoduje to mnogość koniecznych do przeprowadzenia operacji na wielomianach, co bezpośrednio przekłada się czas znajdowania pierwiastków w tym przypadku.



Rysunek 4.7: Czas znajdowania pierwiastków dla wielomianów zawierających kolejne pierwiastki całkowite

Obserwując powyższy wykres trzeba zauważyć wielkie przełożenie stopnia wielomianu na czas działania algorytmu. Wraz z podwojeniem się stopnia testowanego wielomianu widzimy ponad dziesięciokrotny wzrost czasu działania. Tempo wzrostu powyższej funkcji przekracza wzrost sześcienny. Oznacza to, że w przypadku tej klasy wielomianów, której pierwiastkami są kolejne liczby całkowite, mówimy o złożoności wielomianowej.

Niewątpliwy wpływ na wysokie tempo wzrostu ma także wartość kolejnych pierwiastków, która wzrasta wraz ze stopniem wielomianu. Powoduje to, że jesteśmy zmuszeni wykonywać działania na coraz większy liczbach całkowitych. Niektóre z nich posiadają znacznie więcej niż 100 cyfr, co pomimo użycia biblioteki mpir musi mieć znaczący wpływ na czas wykonywania operacji na liczbach.

Wielomiany o pierwiastkach wymiernych

Zachowanie wielomianów może być inne, jeżeli pierwiastki będą się znajdować blisko siebie. By zapewnić im dostatecznie bliskie sąsiedztwo, nie wystarczy by było one liczbami całkowitymi, ale potrzeba by były to liczby wymierne. Podobnie jak w poprzednim przypadku przeprowadzone testy będą zakładały maksymalny stopień wielomianu równy 32. Dobierzmy odległość pomiedzy pierwiastkami tak, by zapewnić, że odległość między dowolnymi z nich była mniejsza od 1. Ustalmy, więc ostęp pomiędzy kolejnymi z nich na wartość 0.01.

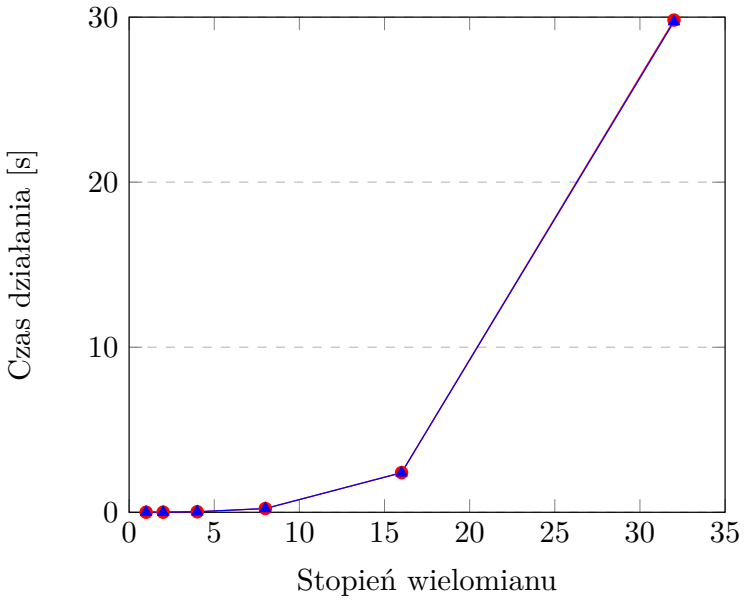
Według powyższych założeń testowany wielomian przyjmie następującą wartość:
$W(x) = (x - 1.01) * (x - 1.02) * (x - 1.03) * ... * (x - (1 + \frac{k-2}{100})) * (x - (1 + \frac{k-1}{100})) * (x - (1 + \frac{k}{100}))$,

gdzie k jest stopniem wielomianu.

Zapoznajmy się z wynikami, by sprawdzić, jak uzyskane wyniki różnią się od wcześniejszego przypadku, w którym pierwiastkami były kolejne liczby naturalne.

Testowany wielomian	Czas dla PolynomialMap [s]	Czas dla PolynomialVector [s]	Współczynnik czasów
$W(x) = x - 0.01$	0.003	0.003	1
$W(x) = (x - 0.01)(x - 0.02)$	0.008	0.008	1
$W(x) = (x - 0.01)(x - 0.02)(x - 0.03)(x - 0.04)$	0.033	0.033	1
$W(x) = (x - 0.01)(x - 0.02) * ... * (x - 0.07)(x - 0.08)$	0.229	0.228	1
$W(x) = (x - 0.01)(x - 0.02) * ... * (x - 0.15)(x - 0.16)$	9.275	9.291	1.002
$W(x) = (x - 0.01)(x - 0.02) * ... * (x - 0.31)(x - 0.32)$	1226.351	1226.333	1

Tabela 4.6: Porównanie czasów znajdowania pierwiastków dla wielomianów zawierających pierwiastki wymierne różniące się o 0.01



Rysunek 4.8: Czas znajdowania pierwiastków dla wielomianów zawierających pierwiastki wymierne różniące się o 0.01

Podobnie jak dla wszystkich innych wielomianów gęstych także w tym przypadku w wyniku porównania struktur otrzymujemy bliźniacze rezultaty. Jednocześnie należy zwrócić uwagę, że wartość pierwiastków wielomianu ma bardzo duże przełożenie na czas ich znajdowania. Widzimy, że w przypadku blisko siebie położonych pierwiastków o niewielkich wartościach zmiennoprzecinkowych czas działania algorytmu jest kilkadziesiąt razy większy niż dla kolejnych liczb naturalnych. Kluczowe wydają się w tym przypadku dwa aspekty. Pierwszym z nich jest konieczność pracy na liczbach niecałkowitych, co powoduje znaczną komplikację działań. Drugim z nich może być wspomniana wcześniej bliskość kolejnych pierwiastków, która powoduje, że pomimo znacznego zawężania przedziału wyszukiwań, w danym zakresie wciąż może znajdować się w nim więcej niż jeden pierwiastek.

4.3 Porównanie aplikacji z innym programem

W niniejszym podrozdziale dokonam porównania stworzonej przeze mnie aplikacji z innym programem. Wybrany przeze mnie został Wolfram Alpha jako przykład serwisu, który oprócz wielu innych funkcjonalności, umożliwia także znajdowanie pierwiastków wielomianów. Porównanie wykonałem w dwóch dziedzinach. Pierwszą z nich jest ocena interfejsu i możliwości jakie dają nam obie aplikacje. Drugą z nich jest wykonanie szeregu testów, mające na celu sprawdzenie specyficznych przypadków wielomianów, by przekonać się, czy są one wystarczająco precyzyjne.

4.3.1 Porównanie możliwości aplikacji i interfejsu użytkownika

Dokonałem porównania obu aplikacji w trzech kategoriach. Pierwszą z nich była możliwość znajdowania pierwiastków. Obie aplikacje umożliwiają lokalizację pierwiastków rzeczywistych, jednak tylko serwis Wolfram Alpha informuje także o wartościach pierwiastków zespolonych. Tę informację można traktować jednak bardziej jako ciekawostkę, gdyż element ten nie był przedmiotem niniejszej pracy.

Drugą kategorią była ocena interejsu pod kątem wprowadzania wielomianów wejściowych w dowolny sposób. Obie aplikacje umożliwiają podawania zarówno kolejnych wyrazów wielomianu w postaci sumy, jak i jego reprezentację w postaci iloczynu. Druga funkcjonalność jest niezwykle przydatna w celach testowych. Umożliwia ona bowiem jawną specyfikę występujących w wielomianie pierwiastków i sprawdzenie, czy rzeczywiście zostaną one zwrócone przez daną aplikację.

Ostatnia ze sprawdzanych kategorii nawiązywała do udogodnień i optymalizacji danej aplikacji. Na przypomnienie zasługuje wspomniany wcześniej fakt, że każdy wielomian można przedstawić w postaci iloczynu wielomianów. Jeżeli szukamy w takim przypadku pierwiastków wystarczy uruchomić algorytm ich znajdowania osobno dla wszystkich wielomianów, a następnie scalić otrzymane rezultaty. Takie działanie z pewnością spowodowałoby znaczny wzrost wydajności działania aplikacji, ponieważ ograniczyłoby to stopień badanych wielomianów, który w wielu przypadkach ma kluczowych wpływ na jej wydajonść.

Aplikacja Wolfram Alpha daje możliwość przedstawienia znalezionych pierwiastków w formie

graficznej przy pomocy wykresu. Dodatkowo należy podkreślić fakt, że serwis ten wykorzystuje moc obliczeniową innych komputerów. Wszystkie działania wykonują się poza sprzętem użytkownika, który na końcu dostaje odpowiedź na zapytanie, które zdefiniował. Na korzyść drugiego programu przemawia także fakt, że jest to serwis internetowy, dzięki czemu użytkownik nie musi przejmować się koniecznością pobierania i instalowania aplikacji, a może z niej skorzystać na dowolnym urządzaniu. Tej wygody nie daje mój program, który wymusza na użytkowniku instalację aplikacji i ściśle definiuje system operacyjny, którym jest 64-bitowy system Windows.

Poniżej zamieszczam podsumowanie porównania obu aplikacji w formie tabeli.

Kategoria	Kryterium	Moja aplikacja	Wolfram Alpha
Znajdowanie pierwiasktów	Pierwiastki rzeczywiste	+	+
	Pierwiastki zespolone	–	+
Wprowadzanie wejścia	W postaci sumy kolejnych wyrazów	+	+
	W postaci iloczynu wielomianów	+	+
Udogodnienia i optymalizacja	Osobne szukanie pierwiastków dla każdego z czynników	–	–
	Rysowanie wykresu	–	+
	Wykorzystanie mocy obliczeniowej innych komputerów	–	+
	Przenośność i brak konieczności instalacji aplikacji	–	+

Tabela 4.7: Porównanie możliwości aplikacji i interefejsu użytkownika

4.3.2 Porównanie poprawności i precyzji obliczeń

W ramach sprawdzania poprawności działania obu aplikacji, postanowiłem przeprowadzić różnorodne testy. Na podstawie przeprowadzonych badań i obserwacji zauważyłem, że kluczowe dla poprawnego działania aplikacji jest to, czy pierwiastki danego wielomianu znajdują się blisko siebie. Jak zostało wspomniane wcześniej, algorytm ciągu Sturma pozwala na znalezienie wszystkich pierwiastków, niezależnie od tego, w jakim położeniu od siebie się znajdują. Aspekt ten daje przekonanie, że stworzona przeze mnie aplikacja, powinna zachowywać się poprawnie. Tezę tę jednak trzeba było zweryfkować, a dodatkowo zbadać działanie programu Wolfram Alpha. Postanowiłem podzielić przeprowadzone testy na dwie grupy. Pierwsza z nich zakłada, że pierwiastki wielomianów znacząco się od siebie różnią. Z kolei druga, sprawdza odporność dla analogicznych testów, ale gdy pierwiastki znajdują się w niewielkiej odległości od siebie.

Pierwiastki odległe od siebie

Dla wartości całkowitych przeprowadzono trzy testy:

- $W(x) = (x - 10^{20})(x - 10^{30})(x - 10^{40})$
- $W(x) = (x - 10)(x - 20)(x - 30) * ... * (x - 80)(x - 90)(x - 100)$
- $W(x) = (x - 10)(x - 20)(x - 30) * ... * (x - 180)(x - 190)(x - 200)$

W przypadku mojego programu wszystkie testy zakończyły się powodzeniem. Dla aplikacji Wolfram Alpha wykonały się poprawnie tylko pierwsze dwa. Ostatni zwrócił komunikat o przekroczeniu limitu czasu w wersji darmowej.

Kolejna kategoria to testowanie wysokich wartości liczbowych dla wielomianów bardzo niskich stopni, tj. odpowiednio – pierwszego, drugiego i trzeciego. Zostały skonstruowane następujące wielomiany testowe.

- $W(x) = (x - 10^{20} - 0.1)$
- $W(x) = (x - 10^{20} - 0.1)(x - 10^{30} - 0.1)$
- $W(x) = (x - 10^{20} - 0.1)(x - 10^{30} - 0.1)(x - 10^{40} - 0.1)$

Moja aplikacja skutecznie poradziła sobie z każdym przypadkiem. Z kolei drugi program zwrócił następujące wyniki.

- $\{10^{20}\}$
- $\{10^{20}, 100000000000000000198846248386 56\}$
- $\{10^{20}, 999999999999999879147136483328, 100000000000000000303786028427003666890752\}$

Można zauważyć, że program nie znalazł dokładnej wartości żadnego z pierwiastków. Dla najmniejszej liczby, równej $10^{20} + 0.1$, wynik nie zawierał części ułamkowej. W pozostałych przypadkach dało się zauważyć duży błąd bezwzględny, tzn. różnicę pomiędzy wartością uzyskaną a dokładną, przy jednocześnie bliskiej zeru wartości błędu względnego, obliczanego jako wartość bezwzględną z ilorazu błędu względnego i wartości dokładnej.

Następnymi trzema kategoriami jest testowanie niewielkich pierwiastków rzeczywistych, występujących w wielomianach stopni: czwartego, szóstego i ósmego. Kategorie te są analogiczne. Jedyną różnicą jest to, że ułamkowa część liczby, jest coraz bardziej dokładna.

Oto wartości wielomianów testowane w kolejnych kategoriach.

1. Niewielkie wartości rzeczywiste, 1 cyfra po przecinku
   - $W(x) = (x - 1.1)(x - 2.2)(x - 3.3)(x - 4.4)$
   - $W(x) = (x - 1.1)(x - 2.2) * ... * (x - 5.5)(x - 6.6)$
   - $W(x) = (x - 1.1)(x - 2.2) * ... * (x - 7.7)(x - 8.8)$
2. Niewielkie wartości rzeczywiste, 2 cyfry po przecinku
   - $W(x) = (x - 1.11)(x - 2.22)(x - 3.33)(x - 4.44)$

- $W(x) = (x - 1.11)(x - 2.22) * ... * (x - 5.55)(x - 6.66)$
- $W(x) = (x - 1.11)(x - 2.22) * ... * (x - 7.77)(x - 8.88)$

3. Niewielkie wartości rzeczywiste, 3 cyfry po przecinku

- $W(x) = (x - 1.111)(x - 2.222)(x - 3.333)(x - 4.444)$
- $W(x) = (x - 1.111)(x - 2.222) * ... * (x - 5.555)(x - 6.666)$
- $W(x) = (x - 1.111)(x - 2.222) * ... * (x - 7.777)(x - 8.888)$

Wszystkie testy dla obu aplikacji zostały zakończone sukcesem. Otrzymane pierwiastki były za każdym razem dokładnie takie, jakie były spodziewane.

Poniżej przedstawiam tabelę, w której umieściłem otrzymane wyniki testów.

Kategoria	Kryterium	Moja aplikacja	Wolfram Alpha
Wartości całkowite	Wielomian 3 stopnia o dużych wartościach pierwiastków	+	+
	Wielomian 10 stopnia	+	+
	Wielomian 20 stopnia	+	–*
Duże wartości rzeczywiste, 1 cyfra po przecinku	Wielomian 1 stopnia	+	+/–
	Wielomian 2 stopnia	+	+/–
	Wielomian 3 stopnia	+	+/–
Niewielkie wartości rzeczywiste, 1 cyfra po przecinku	Wielomian 4 stopnia	+	+
	Wielomian 6 stopnia	+	+
	Wielomian 8 stopnia	+	+
Niewielkie wartości rzeczywiste, 2 cyfry po przecinku	Wielomian 4 stopnia	+	+
	Wielomian 6 stopnia	+	+
	Wielomian 8 stopnia	+	+
Niewielkie wartości rzeczywiste, 3 cyfry po przecinku	Wielomian 4 stopnia	+	+
	Wielomian 6 stopnia	+	+
	Wielomian 8 stopnia	+	+

Tabela 4.8: Porównanie precyzji aplikacji w przypadku odległych od siebie pierwiastków

Pierwiastki znajdujące się blisko siebie

Dla wartości całkowitych przeprowadzono trzy testy:

- $W(x) = (x - 10^{20})(x - 10^{20} - 1)(x - 10^{20} - 2)$
- $(x - 1)(x - 2)(x - 3) * ... * (x - 8)(x - 9)(x - 10)$
- $(x - 1)(x - 2)(x - 3) * ... * (x - 18)(x - 19)(x - 20)$

Poprawność otrzymanych wyników jest taka sama jak w przypadku poprzednich testów dla analogicznej kategorii. Moja aplikacja uzyskała poprawne wyniki dla wszystkich testów, a Wolfram Alpha tylko dla dwóch pierwszych. W trzecim teście zwrócił on komunikat o przekroczeniu limitu czasu i informację o konieczności kupna wersji płatnej, jeżeli chcę uzyskać wynik dla tego przypadku.

Poniżej zamieszczam stworzone testy dla drugiej kategorii. Przedstawione zostały wielomiany stopnii – pierwszego, drugiego i trzeciego, przy czym kolejne pierwiastki różnią się od siebie o 0.1.

- $W(x) = (x - 10^{20} - 0.1)$
- $W(x) = (x - 10^{20} - 0.1)(x - 10^{20} - 0.2)$
- $W(x) = (x - 10^{20} - 0.1)(x - 10^{20} - 0.2)(x - 10^{20} - 0.3)$

Mój program także dla powyższych testów zachował się bezbłędnie. Odmienne zachowanie można zaobserwować dla serwisu Wolfram Alpha. W pierwszym przypadku znalazł on jeden pierwiastek, ale jego wartość zaokrąglił do najbliższej liczby całkowitej. W drugim przypadku program znalazł także tylko jeden pierwiastek. Prawdopodobnie spowodowane zostało to zaokrągleniami, po których wartości pierwiastków były takie same, więc zostały potraktowane przez program jako pierwiastek wielokrotny. W ostatnim przypadku serwis Wolfram Alpha zwrócił zupełnie niepoprawny rezultat. Znalazł tylko jeden pierwiastek rzeczywisty, ale jego wartość była zauważalnie różniąca się od wartości dokładnej, przy stosunkowo niewielkim błędzie bezwzględnym. Pozostałych dwóch pierwiastków rzeczywistych Wolfram Alpha nie znalazł. Mamy taką pewność, dzięki temu, że informuje on też użytkownika o ewentualnych pierwiastkach zespolonych. Oba z nich zostały potraktowane jako pierwiastki zespolone, o przeciwnych współczynnikach przy jednostce urojonej i. Oto wartości zwrócone przez aplikację.

- $\{10^{20}\}$
- $\{10^{20}\}$
- $\{x = 99999359596691898368\}, \{10^{20} - 5.54607 * 10^{14}i, 10^{20} + 5.54607 * 10^{14}i\}$

Ostatnie trzy kategorie wyglądają podobnie jak w przypadku testów dla pierwiastków znacznie od siebie odległych. Podobnie jak tam, zostały przeprowadzone testy dla wielomianów stopni: czwartego, szóstego i ósmego. W kolejnych kategoriach wartości pierwiastków są sobie coraz bliższe i różnią się odpowiednio o 0.1, 0.01 i 0.001. Oto jak wyglądają poszczególne wielomiany w przedstawionych wyżej kategoriach.

1. Niewielkie wartości rzeczywiste, 1 cyfra po przecinku
    - $W(x) = (x - 1.1)(x - 1.2)(x - 1.3)(x - 1.4)$
    - $W(x) = (x - 1.1)(x - 1.2) * ... * (x - 1.5)(x - 1.6)$
    - $W(x) = (x - 1.1)(x - 1.2) * ... * (x - 1.7)(x - 1.8)$

2. Niewielkie wartości rzeczywiste, 2 cyfry po przecinku
   - $W(x) = (x - 1.11)(x - 1.12)(x - 1.13)(x - 1.14)$
   - $W(x) = (x - 1.11)(x - 1.12) * ... * (x - 1.15)(x - 1.16)$
   - $W(x) = (x - 1.11)(x - 1.12) * ... * (x - 1.17)(x - 1.18)$
3. Niewielkie wartości rzeczywiste, 3 cyfry po przecinku
   - $W(x) = (x - 1.111)(x - 1.112)(x - 1.113)(x - 1.114)$
   - $W(x) = (x - 1.111)(x - 1.112) * ... * (x - 1.115)(x - 1.116)$
   - $W(x) = (x - 1.111)(x - 1.112) * ... * (x - 1.117)(x - 1.118)$

W pierwszej z powyższych kategorii obie aplikacji znalazły wszystkie pierwiastki rzeczywiste z wymaganą dokładnością.

Dla drugiej kategorii mój program także znalazł wszystkie rozwiązania. Z kolei Wolfram Alpha zachował się poprawnie dla wielomianów stopni czwartego i szóstego. Wynik znajdowania pierwiastków dla ostatniego wielomianu, stopnia ósmego był niepoprawny. Znalezione zostały tylko dwa pierwiastki rzeczywiste, przy czym w obu przypadkach widoczna była pewna niedokładność. Pozostałe sześć pierwiastków rzeczywistych nie zostało znalezionych, a wszystkie one zostały potraktowane jako pierwiastki zespolone. Poniżej zamieszczam zwrócone przez serwis Wolfram Alpha wartości pierwiastków.

- $\{1.10922, 1.18099\}, \{1.12238 - 0.00620719i, 1.12238 + 0.00620719i,$
  $1.14483 - 0.0113488i, 1.14483 + 0.0113488i, 1.16769 - 0.00684386i, 1.16769 + 0.00684386i\}$

Także dla trzeciej kategorii, wymagającej najwyższej precyzji, moja aplikacja zachowała się w pełni poprawnie. Z kolei porównywany program tylko dla wielomianu stopnia czwartego zwrócił poprawny wynik. W drugim przypadku wielomian Wolfram Alpha znalazł 2 pierwiastki rzeczywiste. Oba cechowały się zauważalnym brakiem precyzji. W ostatnim przypadku, dla wielomianu stopnia ósmego Wolfram Alpha nie znalazł żadnego pierwiastka rzeczywistego. Oto wartości pierwiastków zwrócone przez porównywany serwis dla drugiego i trzeciego wielomianu tej kategorii.

- $\{1.10875, 1.11827\}, \{1.11113 - 0.0035388i, 1.11113 + 0.0035388i,$
  $1.11587 - 0.00355076i, 1.11587 + 0.00355076i\}$
- $\{\}, \{1.09738 - 0.0069297i, 1.09738 + 0.0069297i, 1.10729 - 0.016855i,$
  $1.10729 + 0.016855i, 1.12154 - 0.0170316i, 1.12154 + 0.0170316i, 1.13179 - 0.00710625i,$
  $1.13179 + 0.00710625i\}$

Reasumując przeprowadzone testy należy zauważyć, że moja aplikacja zachowała się poprawnie w przypadku wszystkich przeprowadzonych testów. Kluczowe były dla tego dwa aspekty. Pierwszym z nim była dowolna precyzja obliczeń, którą zapewniała biblioteka mpir. Drugim zaś było zastosowanie twierdzenia Sturma, umożliwiającego znalezienie wszystkich pierwiastków rzeczywistych.

Z kolei aplikacja Wolfram Alpha miała spore problemy ze znalezieniem pierwiastków, w momencie, gdy znajdowały się one blisko siebie. Spowodowane było to najprawdopodobniej zaokrągleniami, które zostały wykonywane w trakcie obliczeń. Sprawiało to, że czasem wartości pierwiastków były zauważalnie niedokładne, co momentami przekładało się na traktowanie różnych pierwiastków jako jeden pierwiastek wielokrotny. Dodatkowo, zwłaszcza przy wzroście wymaganej precyzji obliczeń i stopnia badanych wielomianów, dało się zauważyć, że aplikacja Wolfram Alpha nie potrafi zlokalizować pierwiastków danego wielomianu.

Poniżej znajduje się tabela z podsumowaniem przeprowadzonych testów.

Kategoria	Kryterium	Moja aplikacja	Wolfram Alpha
Wartości całkowite	Wielomian 3 stopnia o dużych wartościach pierwiastków	+	+
	Wielomian 10 stopnia	+	+
	Wielomian 20 stopnia	+	–*
Duże wartości rzeczywiste, 1 cyfra po przecinku	Wielomian 1 stopnia	+	+/–
	Wielomian 2 stopnia	+	–
	Wielomian 3 stopnia	+	–
Niewielkie wartości rzeczywiste, 1 cyfra po przecinku	Wielomian 4 stopnia	+	+
	Wielomian 6 stopnia	+	+
	Wielomian 8 stopnia	+	+
Niewielkie wartości rzeczywiste, 2 cyfry po przecinku	Wielomian 4 stopnia	+	+
	Wielomian 6 stopnia	+	+
	Wielomian 8 stopnia	+	–
Niewielkie wartości rzeczywiste, 3 cyfry po przecinku	Wielomian 4 stopnia	+	+
	Wielomian 6 stopnia	+	–
	Wielomian 8 stopnia	+	–

Tabela 4.9: Porównanie precyzji aplikacji w przypadku położonych blisko siebie pierwiastków

Rozdział 5

Podsumowanie

Celem pracy była implementacja algorytmu, pozwalającego na obliczanie pierwiastków rzeczywistych dowolnego wielomianu. Dokonałem tego, korzystając z twierdzenia Sturma. Przeprowadziłem analizę, jaka repzentacja wielomanu jest bardziej wydajna. Dodatkowo, porównałem także możliwości stwrzonej przez mnie aplikacji z serwisem Wolfram Alpha, który umożliwia m. in. obliczanie pierwiastków rzeczywistych wielomianów.

Stworzona przeze mnie aplikacja została podzielona na trzym moduły. Są to biblioteka statyczna, framework testowy oraz aplikacja konsola. Pierwsza z nich zawiera całą logikę aplikacji, pozwala wykonywać działania na wielomianach, korzystając z biblioteki mpir. Ta ostatnia umożliwia działania na dużych liczbach o wysokiej precyzji, przy pomocy zoptymalizowanego kodu asemblerowego, co sprawia, że przeprowadzone operacje są maksymalnie wydajne. Drugą część stanowi framework testowy, który w narzedziu Microsoft Visual Studio 2015 pozwala na definiowanie i egzekucję testów jednostkowych. W moim przypadku poza prostymi testami zostały w nich zdefiniowane także bardziej skomplikowane testy funkcjonalne. Trzecim modułem jest aplikacja konsolowa, dzięki której użytkownik może wprowadzać wielomiany oraz przedział, w którym chcemy znaleźć wszystkie pierwiastki rzeczywiste z podaną precyzją.

Implementacja wielomianu została oparta na klasie abstrakcyjnej Polynomial. Na jej podstawie zostały wykonane dwie klasy, różniące się reprezentacją wielomianu na której bazują. Pierwszą z nich została klasa oparta na mapie, co pozwalało na posiadanie informacji wyłącznie o niezerowych współczynnikach. Z kolei drugą z nich jest tablica, wymuszająca reprezentację także zerowych współczynników wielomianów.

Przeprowadziłem testy wydajnościowe porównujące czas działania algorytmu dla obu klas. Otrzymane wyniki były niemal identyczne dla obu typów wielomianów w przypadku testowania wielomianów gęstych. Z kolei dla wielomianów rzadkich widoczna była bardzo duża dysproporcja na korzyść PolynomialMap, która wraz ze wzrostem stopnia testowanego wielomianu ulegała jeszcze powiększeniu. Przeprowadzone testy wykazały, że znajdowanie pierwiastków rzeczywistych wielomianu jest najszybsze dla rzadkich wielomianów, zawierających niewielką liczbę pierwiastków rzeczywistych. Wzrost liczby niezerowych współczynników i liczby pierwiastków rzeczywistych

powoduje znaczne wydłużenie się czasu działania algorytmu dla obu struktur. Maksymalną wartość osiąga on dla przypadku, kiedy mamy do czynienia z gęstym wielomianem, zawierającym wyłącznie pierwiastki rzeczywiste. Dodatkowo powinny one znajdować się blisko siebie i zawierać skomplikowaną część ułamkową.

Dokonałem porównania możliwości wytworzonej przeze mnie aplikacji, z innym programem, który umożliwia znajdowanie pierwiastków wielomianów. Wybrałem serwis Wolfram Alpha. Porównałem interfejsy i możliwości obu aplikacji. Z przedstawionego zestawienia wynikało, że różnice są niewielkie, ale serwis Wolfram Alpha posiada kilka dodatkowych funkcji takich jak rysowanie wykresu, czy znajdowanie pierwiastków zespolonych, których nie posiada moja aplikacja.

Po dogłębnej analizie okazało się, że dla pewnych wielomianów Wolfram Alpha dokonuje zaokrągleń, które powodują pewne niedokładności w wartości znalezionych pierwiastków lub zupełnie uniemożliwiają ich znalezienie. Dzieje się tak, gdyż wartości współczynników wielomianów są w pewnych okolicznościach zaokrąglane. Fakt ten pokazuje, jak ważnym aspektem pracy było znalezienie sposóbu na reprezentację wartości liczbowych, dowolnej precyzji. Bez tego elementu skuteczne znajdowanie pierwiastków wielomianów jest zazwyczaj niemożliwe i działa tylko dla bardzo prostych przypadków.

Spis rysunków

Spis tabel

Bibliografia

[1] E.J. Barbeau. *Polynomials*. Problem Books in Mathematics. Springer New York, 2003.

[2] D. Buell. *Algorithmic Number Theory: 6th International Symposium, ANTS-VI, Burlington, VT, USA, June 13-18, 2004, Proceedings*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004.

[3] R.L. Burden, J.D. Faires, and A.M. Burden. *Numerical Analysis*. Cengage Learning, 2015.

[4] L.N. Childs. *A Concrete Introduction to Higher Algebra*. Undergraduate Texts in Mathematics. Springer New York, 2012.

[5] T.H. Cormen and K. Diks. *Wprowadzenie do algorytmów*. Klasyka Informatyki. Wydawnictwa Naukowo-Techniczne, 2004.

[6] T. Granlund and G.D. Team. *Gnu MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, 2015.

[7] W. Kryszewski. *Wykład analizy matematycznej, cz. 1: Funkcje jednej zmiennej:*. Number pkt 1. Wydawnictwo Naukowe UMK, 2014.

[8] D.S. Malik. *Data Structures Using C++*. Cengage Learning, 2009.

[9] Polskie Towarzystwo Matematyczne. *Wiadomości matematyczne*. Number t. 10-11. Państwowe Wydawn. Naukowe., 1968.

[10] J.M. McNamee. *Numerical Methods for Roots of Polynomials -*. Number pkt 1 in Studies in Computational Mathematics. Elsevier Science, 2007.

[11] T. Mora. *Solving Polynomial Equation Systems I: The Kronecker-Duval Philosophy*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003.

[12] V. Pan. *Structured Matrices and Polynomials: Unified Superfast Algorithms*. Birkhäuser Boston, 2012.

[13] W. Sierpiński and A. Mostowski. *Zasady algebry wyższej, z przypisem Andrzeja Mostowskiego Zarys teorii Galois*. Monografie Matematyczne. Nakładem Polskiego Tow. Matematycznego, 1951.

[14] Mieczysław (1918-2007) Warmus and Józef (1927-) Łukaszewicz. Metody numeryczne i graficzne. cz. 1.