10.05.2019r. Wrocław

Organizacja i architektura komputerów

INEK00022P

Projekt

Wydział Elektroniki	**Kierunek:** Informatyka
Grupa zajęciowa: Czwartek TP 13 – 15	**Semestr:** 2018/19 Lato
Imię i nazwisko: Jan Miszczyk, Paweł Parczyk	**Nr albumu**: 241366, 241390
Prowadzący: Dr inż. Tadeusz Tomczak	

Temat:

Implementacja biblioteki liczb zmiennoprzecinkowych dowolnej precyzji

Ocena:

Data: 23.05.2019r.

1. Cel projektu

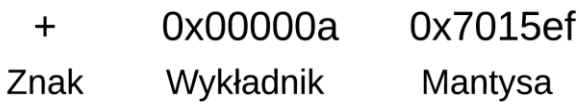
Celem projektu była implementacja podstawowych operacji arytmetycznych dla liczb zmiennoprzecinkowych zapisanych zgodnie ze standardem **IEEE 754**. Nasza biblioteka pozwala użytkownikowi na ustalenie precyzji działań, poprzez możliwość zmiany ilości bitów z których składa się mantysa oraz wykładnik. W celu minimalizacji zużycia pamięci, użyte struktury danych składają się z bajtów (pojedynczy bit nie jest przechowywany jako typ `bool`). Taka reprezentacja danych wymagała od nas stworzenia metod, które pozwalają na swobodną manipulację bitową kontenerów opisujących dane wielkości. Projekt został zrealizowany w języku **C++** z wykorzystaniem szablonów oraz biblioteki **STL**.

Operacje arytmetyczne, wspierane przez naszą implementację to kolejno: dodawanie, odejmowanie, mnożenie, dzielenie oraz wyznaczanie pierwiastka kwadratowego. Dostępna jest jedna metoda zaokrąglania wyników – symetrycznie do parzystej (round ties to even). Niestety, nie udało nam się zrealizować wsparcia dla liczb zdenormalizowanych – w naszym programie będą one reprezentowane jako zero. Szczegóły dotyczące wykorzystanych algorytmów i zasad postępowania zostaną wyjaśnione w dalszej części raportu.

2. Realizacja projektu

2.1. Reprezentacja liczby oraz interakcja z użytkownikiem

W celu umożliwienia wyświetlania liczb dużej precyzji, nasza implementacja korzysta z zapisu szesnastkowego, który pozwala zminimalizować ilość znaków potrzebnych do przedstawienia zawartości danego kontenera. Użytkownik widzi wykładnik w postaci kodu U2 (obciążenie wykładnika zostaje odjęte), a mantysę jako ciąg kolejnych bajtów.



Rysunek 1 - przykładowa reprezentacja liczby

Wprowadzanie danych do programu może być zrealizowane na jeden z trzech sposobów:

1. Pobranie wartości liczby pojedynczej precyzji (konstruktor `float`).
2. Pobranie wartości liczby podwójnej precyzji (konstruktor `double`).
3. Wprowadzenie liczby w postaci ciągu znaków – reprezentacja szesnastkowa (osobno wykładnik – bez obciążenia, osobno mantysa).

Konstruktory korzystające z wartości liczbowej `float` oraz `double` zawsze zapisują tyle bitów mantysy, ile jest w stanie pomieścić wybrana poprzez szablon implementacja [1]. Wykładnik jest przekształcany do liczby całkowitej bez znaku, a następnie dodawane jest do niego odpowiednie obciążenie ($2^{k-1} - 1$, gdzie k to liczba bitów wykładnika). Ilość bajtów potrzebnych dla danej reprezentacji jest obliczana według poniższych wzorów i zapisywana z użyciem prywatnego konstruktora klasy [2].

$$size_{exponent} = \frac{bit\ count_{exponent}}{8} + 1$$

$$size_{fraction} = \frac{bit\ count_{fraction}}{8} + 1$$

Wywołaniu konstruktora prywatnego towarzyszy również ustalenie stałych `maxExponent`, `minExponent` oraz `biasContainer`. Upraszczają one późniejsze wykonywanie operacji, takich jak sprawdzenie nadmiaru, czy wyświetlenie wykładnika w postaci bez obciążenia. Konstruktor kopiujący, jak również operatory ułatwiające wprowadzenie oraz wyświetlanie liczb z użyciem strumieni, zostały także zaimplementowane [3].

Dostępna jest również możliwość wyświetlania liczby w postaci binarnej, jednak z racji swojej długości jest ona nieprzystępna dla użytkownika. Liczby specjalne są wyświetlane jako odpowiadające im symbole: `+/- inf, +/- nan`.

---

[1] Konstruktor `VariableFloat(float number)` – plik VariableFloat.h, linia 241.
[2] Konstruktor prywatny `VariableFloat()` – plik VariableFloat.h, linia 207.
[3] Patrz – definicja klasy `VariableFloat`.

2.2. Algorytm dodawania

Algorytm dodawania liczb zmiennoprzecinkowych opiera się na wyrównaniu mantysy liczby mniejszej. W naszej klasie został zdefiniowany z użyciem przeciążonego operatora dodawania. Po odjęciu mniejszego wykładnika od większego, otrzymujemy liczbę przesunięć potrzebną do wyrównania wyżej wspomnianej mantysy. Takie odejmowanie możemy przeprowadzić bezpośrednio na wykładnikach z dodanym obciążeniem. Przeniesienie z najbardziej znaczącego bitu pozwoli nam stwierdzić, który z wykładników jest tym mniejszym [4].

$$e_1 + bias - (e_2 + bias) = e_1 - e_2$$

Po dodaniu z przodu obu mantys „ukrytej jedynki", denormalizujemy liczbę mniejszą, przesuwając bitowo jej ułamek w prawo o obliczoną przez nas ilość bitów. Tak przygotowane kontenery mogą zostać do siebie dodane, a wynik ponownie znormalizowany poprzez znalezienie pozycji najwyższego, ustawionego bitu [5]. „Ukryta jedynka" jest usuwana, a kontenery mantysy oraz wykładnika liczby zwracanej są ustawiane (podczas ustawiania wartości mantysy przeprowadzone jest również zaokrąglenie [6]).

2.3. Algorytm odejmowania

Odejmowanie liczb zostało przez nas zrealizowane z użyciem wcześniej zdefiniowanego operatora dodawania. Najprostszy sposób to zmiana znaku drugiego operandu operacji. Suma tych dwóch argumentów będzie wtedy wynikiem pierwotnego odjęcia.

$$a - b = a + (-b)$$

Wykrywanie nadmiaru, niedomiaru oraz zaokrąglanie mantysy wyniku są częścią metody przeciążonego operatora sumy.

---

[4] Patrz – metoda `subtractBytes` – plik ByteArray.cpp, linia 129.
[5] Metoda `findHighestOrderOnePosition` – plik ByteArray.cpp, linia 314.
[6] Setter pola `fractionContainer` oraz metoda `roundFraction` – plik VariableFloat.h, linia 157 i 751.

2.4. Algorytm mnożenia

Algorytm mnożenia liczb zmiennoprzecinkowych został zaimplementowany z użyciem poniższej zależności:

$$M_1 * 2^{e_1} * M_2 * 2^{e_2} = M_1 * M_2 * 2^{(e_1 + e_2)}$$

Z racji obciążenia wykładników, od jednego z nich musi zostać ono odjęte, aby uniknąć dwukrotnego dodania bias'u. Po doprowadzeniu jednego z wykładników do postaci liczby całkowitej, wykonujemy dodawanie. Wynik mnożenia mantys może być maksymalnie dwa razy większej długości niż dotychczasowy ułamek. Następnym krokiem jest normalizacja liczby. Znajdujemy pozycję najbardziej znaczącego ustawionego bitu wyniku mnożenia i wykonujemy przesunięcie bitowe (usuwając też „ukrytą jedynkę"). Wynikowy wykładnik jest odpowiednio poprawiany. Wykrywanie nadmiaru oraz zaokrąglanie jest zrealizowane w taki sam sposób, jak w poprzednich operacjach.

2.5. Algorytm dzielenia

Dzielenie liczb zmiennoprzecinkowych w naszym projekcie jest zrealizowane z wykorzystaniem algorytmu dzielenia odtwarzającego [7]. Pierwszym krokiem jest wykrycie potencjalnie błędnych operacji, takich jak dzielenie przez zero. Do tego celu służy funkcja publiczna klasy `VariableFloat – isZero()`. Jeśli żaden z operandów nie jest zerem ($0/a$ jest również rozpatrywane podczas poprzedniego kroku), odejmujemy wykładnik drugiego argumentu od wykładnika dzielnej (pamiętając o usunięciu obciążenia). Dzielenie liczb w formacie zmiennoprzecinkowym przedstawia poniższa zależność:

$$M_1 * 2^{e_1} / M_2 * 2^{e_2} = M_1 / M_2 * 2^{(e_1 - e_2)}$$

Po dodaniu bitu „ukrytej jedynki" z przodu obu mantys, wykonujemy algorytm dzielenia odtwarzającego na bajtach. Gdy liczby są tej samej długości oraz każda z nich ma ustawiony

[7] Metoda `divideBytes` – plik ByteArray.cpp, linia 231.

najbardziej znaczący bit, iloraz może być równy jeden lub zero. Korzystając z reszty powstałej po wykonaniu dzielenia, mnożymy ją przez dwa oraz ponownie wykonujemy dzielenie w celu uzyskania kolejnego bitu wyniku. Tak obliczony ciąg bitów (z zadaną jako argument funkcji `divideBytes` dokładnością), może być znormalizowany poprzez znalezienie pozycji najstarszego, ustawionego bitu i przesunięcie kontenera w lewo $index + 1$ razy (usuwamy również ukrytą jedynkę). Na koniec poprawiony zostaje wykładnik wyniku poprzez odjęcie wyliczonej przez nas poprzednio ilości przesunięć. Zaokrąglenie oraz wykrywanie nadmiaru odbywa się podobnie, jak w poprzednich metodach.

2.6. Algorytm obliczania pierwiastka kwadratowego

Pierwiastek kwadratowy jest zaimplementowany z użyciem algorytmu poznanego podczas kursu Architektury Komputerów 1. Pozwala on na obliczenie pierwiastka danej liczby z dowolną dokładnością. Opiera się on na zależności, którą można opisać w poniższy sposób:

$$(2 * Q_i * \beta + X) * X \leq r_i, \text{ gdzie}$$

Q_i – dotychczasowe przybliżenie pierwiastka.

β – baza systemu liczbowego.

X – największa liczba spełniająca podane równanie (1 lub 0 w przypadku systemu binarnego).

r_i – dotychczasowy wynik odjęcia.

Pierwszym krokiem jest, tak jak w dzieleniu, wykrycie błędnych operacji (np. pierwiastek z liczby ujemnej jest nieliczbą). Jeśli dana liczba reprezentuje nieskończoność, to wynikiem operacji pierwiastka powinna być również nieskończoność. Pierwiastki z liczb ujemnych są zdefiniowane w standardzie jako nieliczby, dlatego $-inf$ oraz każda inna liczba z ustawionym bitem $sign$ staje się „NaN'em". Jeśli podany argument jest pierwiastkowalny, sprawdzamy czy jego wykładnik jest parzysty. Gdy okaże się, że cecha danej liczby jest nieparzysta, odejmujemy jedynkę (pamiętając o odpowiednim przesunięciu mantysy). Kiedy wykładnik jest parzysty korzystamy z poniższej tożsamości:

$$\sqrt{M * 2^e} = \sqrt{M} * 2^{e/2}$$

Pierwiastek z ułamka obliczamy za pomocą metody `squareRootBytes`, która implementuje wyżej wymieniony algorytm. Ostatnim krokiem jest normalizacja mantysy. Inne konieczne operacje (nadmiar lub niedomiar) są wykonywane jak w poprzednich funkcjach.

2.7. Zaokrąglanie

Klasa `VariableFloat` posiada jedną metodę zaokrąglania – symetrycznie do parzystej (ang. round ties to even). Wykonuje się za każdym razem, kiedy wywoływany jest setter pola `fractionContainer`. W celu zaokrąglenia mantysy wyznaczamy wartości bitu R (pierwszy po ostatnim bicie ułamka) oraz S (suma logiczna wszystkich pozostałych) [8]. Na podstawie wyznaczonych wartości podejmujemy decyzję:

- R = 0, S = 0 – wartość dokładna, obcinamy ($< 0.5ulp$).
- R = 0, S = 1 – również obcinamy ($< 0.5ulp$).
- R = 1, S = 0 – zaokrąglamy tak, aby najmniej znaczący bit mantysy był parzysty.
- R = 1, S = 1 – zaokrąglamy w górę ($> 0.5ulp$).

Z zaokrąglonej mantysy usuwamy z prawej strony bajty, jeśli ich liczba jest większa od `exponentSize` [9].

2.8. Wartości specjalne

W standardzie IEEE 754 mamy do czynienia z wartościami specjalnymi, które są określone poprzez odpowiednie ustawienie bitów cechy oraz mantysy. Każda z nich (z wyjątkiem liczb zdenormalizowanych), została zaimplementowana w naszym projekcie. Są to kolejno:

- $(+/-)\ 0$ – wszystkie bity są zerami (ustawiony tylko znak dla zera ujemnego).
- $(+/-)\ \infty$ – ustawione wszystkie bity wykładnika, mantysa równa 0.
- NaN – nieliczby, ustawione wszystkie bity wykładnika, mantysa różna od zera.

[8] Metoda `roundFraction()` – plik VariableFloat.h, linia 751.
[9] Patrz – 2.1 Reprezentacja liczby oraz interakcja z użytkownikiem.

2.9. Manipulacja bitowa

Operacje przesunięć bitowych oraz inne funkcje arytmetyczno-logiczne są zdefiniowane dla typów podstawowych. Z racji tego, że nasza biblioteka nie korzysta z kontenerów stałej długości (zdefiniowane ciągi bajtów mogą być dowolnie długie), stworzyliśmy statyczną klasę `ByteArray`, która pozwala na bitową manipulację wektorów bajtów oraz wykonywanie operacji arytmetycznych. Ważniejsze, zaimplementowane przez nas metody są przedstawione poniżej:

`setBit(std::vector<u_char> &array, u_int position, bool value)` – modyfikuje bit znajdujący się na pozycji $position$ kontenera, ustawiając wartość daną jako $value$. Wyznaczenie bajtu oraz pozycji wewnątrz niego odbywa się za pomocą poniższych zależności:

$$byte\ position = position\ /\ 8$$

$$bitPosition = position\ mod\ 8$$

Użycie maski oraz operacji `and` lub `or` (w przypadku, gdy $value$ jest równe 1), pozwala ustawić wybrany bit.

`getBit(std::vector<u_char> &array, u_int position)` – zwraca wartość logiczną bitu znajdującego się na pozycji $position$ kontenera.

`shiftVector[Left/Right](std::vector<u_char> &vector, int shift)` – umożliwia przesunięcie bitowe wektora o $shift$ bitów w lewo lub w prawo.

`getBytesFromInt(u_int value, u_int size)` – zwraca wartość zapisaną w liczbie `u_int`, jako wektor bajtów o długości $size$.

`addBytes()`[10] – dodaje bajty operandu drugiego do pierwszego (wynik w pierwszym). Jeśli wystąpi przeniesienie z najwyższej pozycji, zwraca wartość logiczną $true$.

`subtractBytes()`[10] – odejmuje bajty operandu drugiego od pierwszego (wynik w pierwszym). Jeśli wystąpi przeniesienie z najwyższej pozycji, zwraca $true$.

`multiplyBytes()`[11] – mnoży bajty obu operandów. Rezultat operacji jest przechowywany w pierwszym argumencie. Długość wektora wynikowego może być maksymalnie dwa razy większa od początkowej.

`divideBytes()`[11] – dzieli bajty obu operandów z zadaną prezycją bitową. Wykorzystuje algorytm dzielenia odtwarzającego. Działa tylko dla operandów przedstawiających liczbę znormalizowaną, które mają tą samą długość. Wynik działania jest przechowywany w pierwszym argumencie.

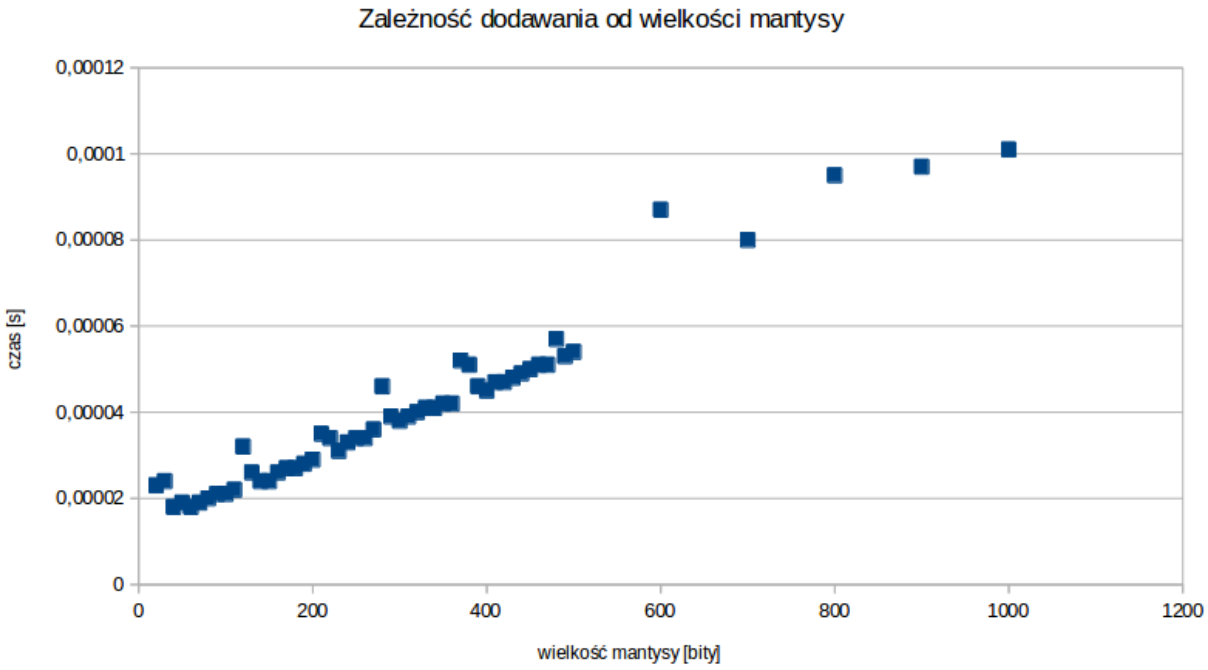
`squareRootBytes()`[11] – oblicza wartość pierwiastka kwadratowego z zadanym przybliżeniem. Korzysta z algorytmu obliczania pierwiastka bit po bicie [12]. Wynik operacji jest przechowywany w pierwszym argumencie wywołania.

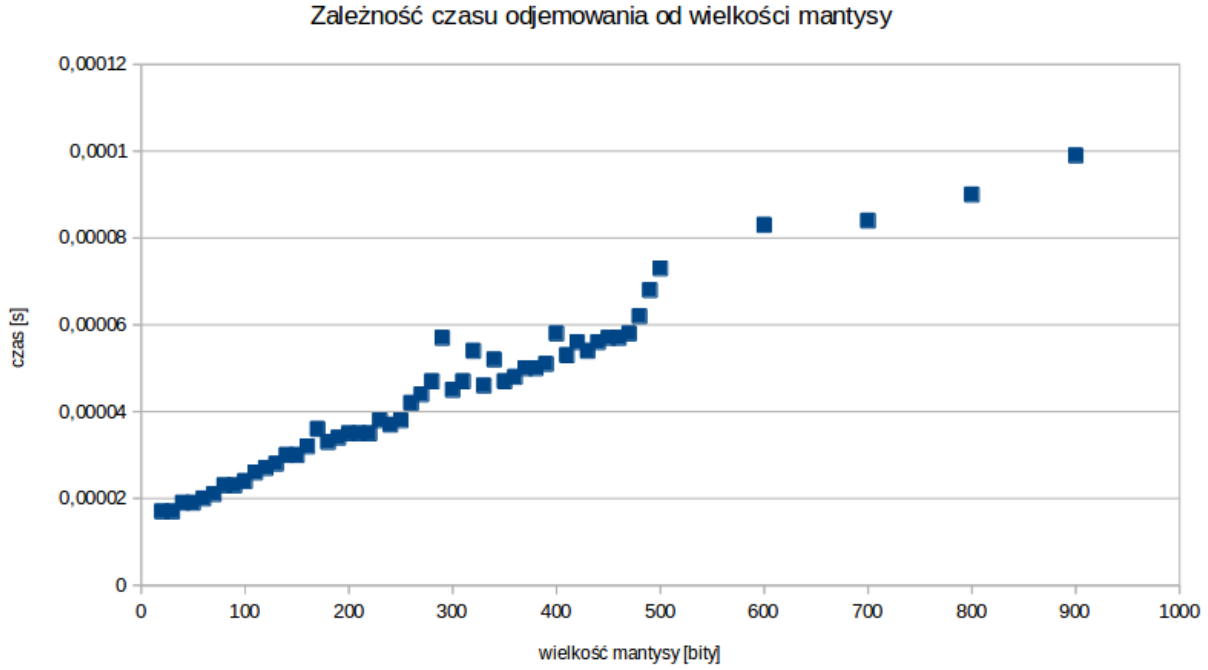
3. Wydajność

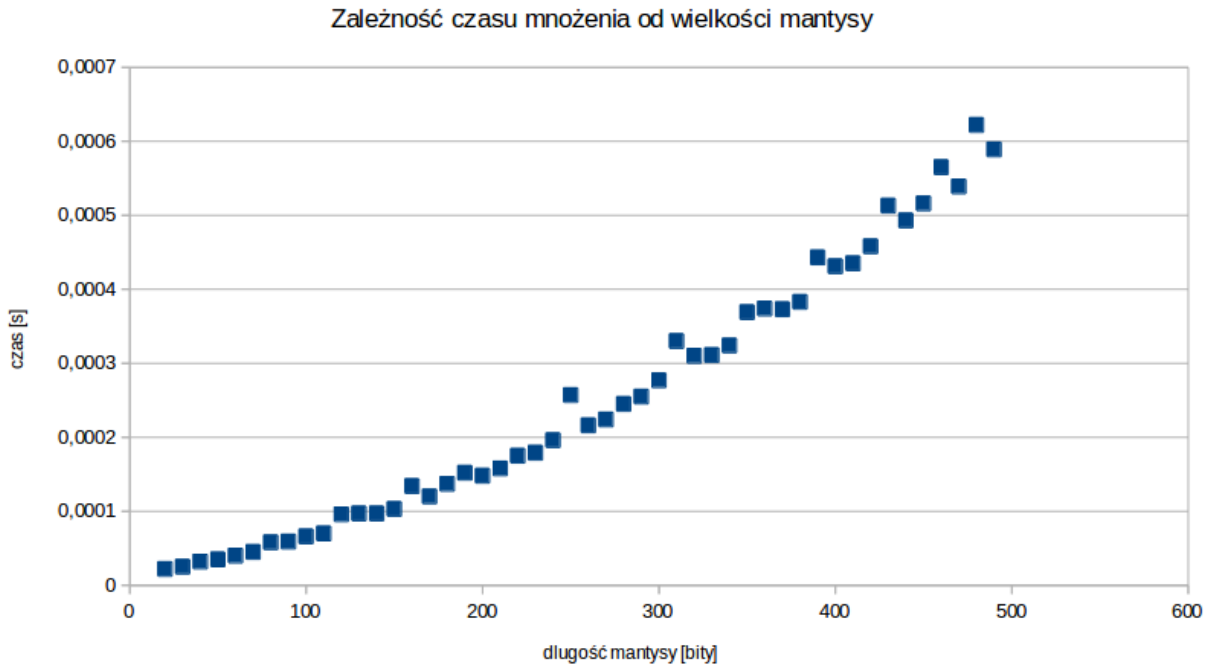
3.1. Testy wydajności

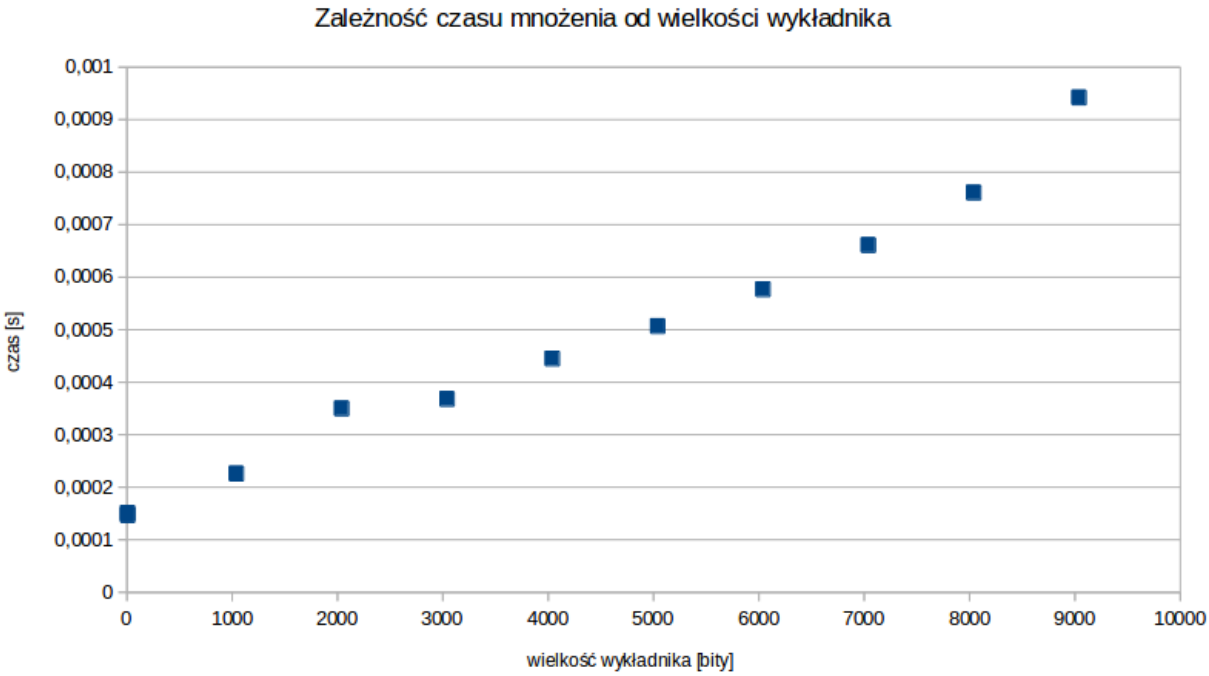
Dla każdej z operacji, generowane jest 40 liczb typu `float`. Testy są przeprowadzane dla zdefiniowanych wielkości mantysy przy stałej ilości bitów wykładnika i odwrotnie. Wynik dla danej wielkości jest średnią czasu wykonania operacji (łącznie wykona się 20 razy). Na następnych stronach znajdują się wykresy opisujące zależności czasowe wykonywania danej operacji od wielkości struktury.

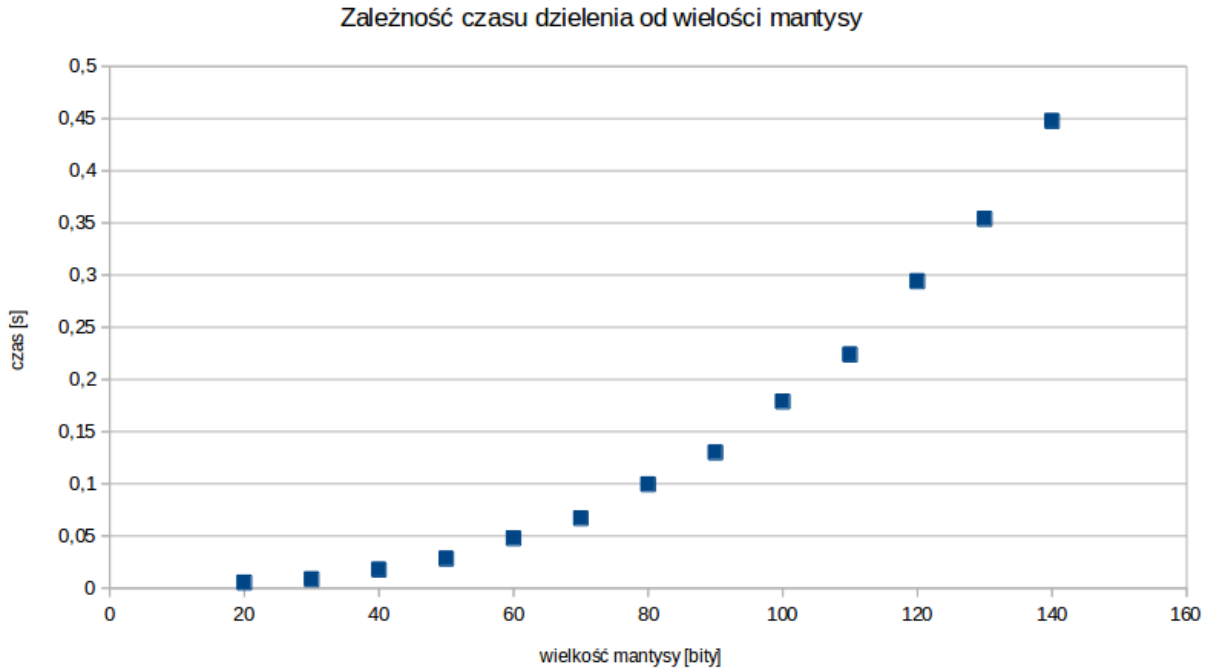
[10] Pełny prototyp w pliku ByteArray.h.
[11] Pełny prototyp w pliku ByteArray.h.
[12] Zależności matematyczne – punkt 2.6 Algorytm obliczania pierwiastka kwadratowego.

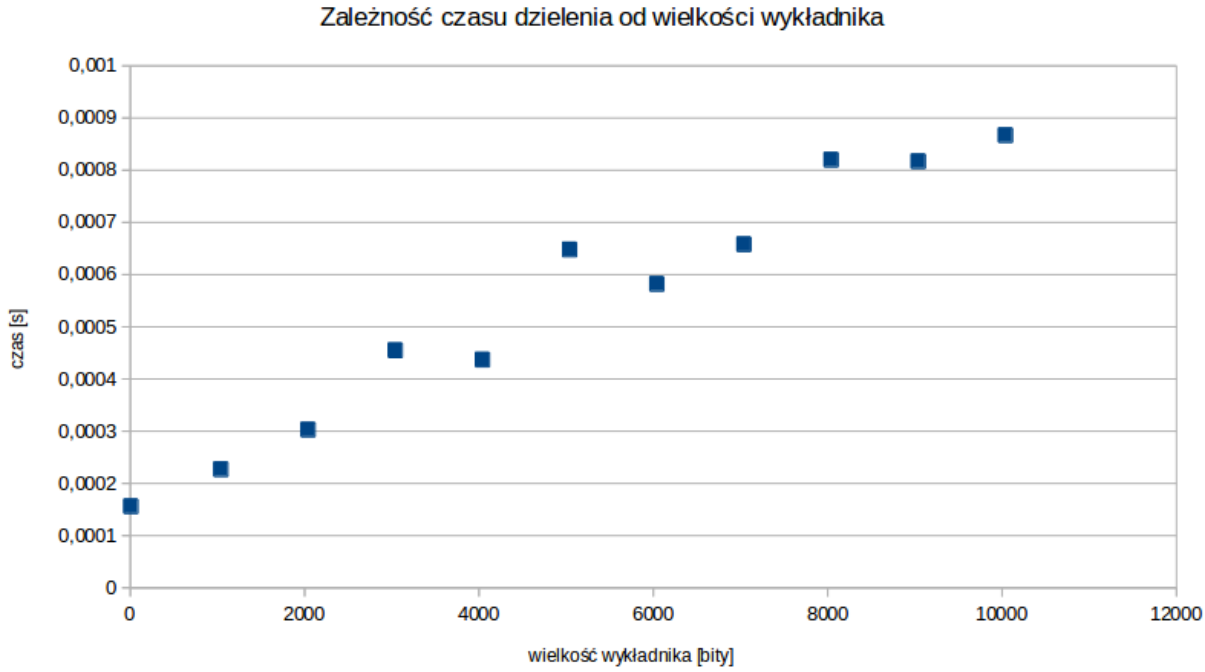

Zależność dodawania od wielkości mantysy
0,00012
0,0001
0,00008
0,00006
0,00004
0,00002
0
czas [s]
0
200
400
600
800
1000
1200
wielkość mantysy [bity]

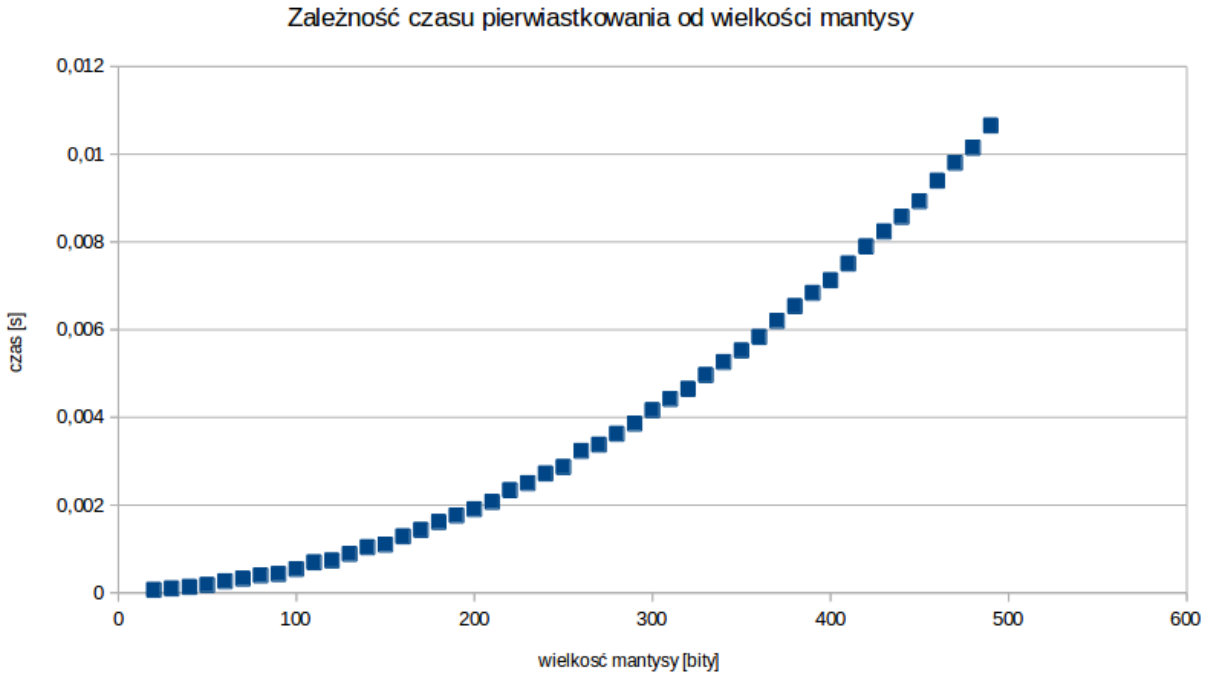

Zależność czasu odjemowania od wielkości mantysy
0,00012
0,0001
0,00008
0,00006
0,00004
0,00002
0
czas [s]
0
100
200
300
400
500
600
700
800
900
1000
wielkość mantysy [bity]

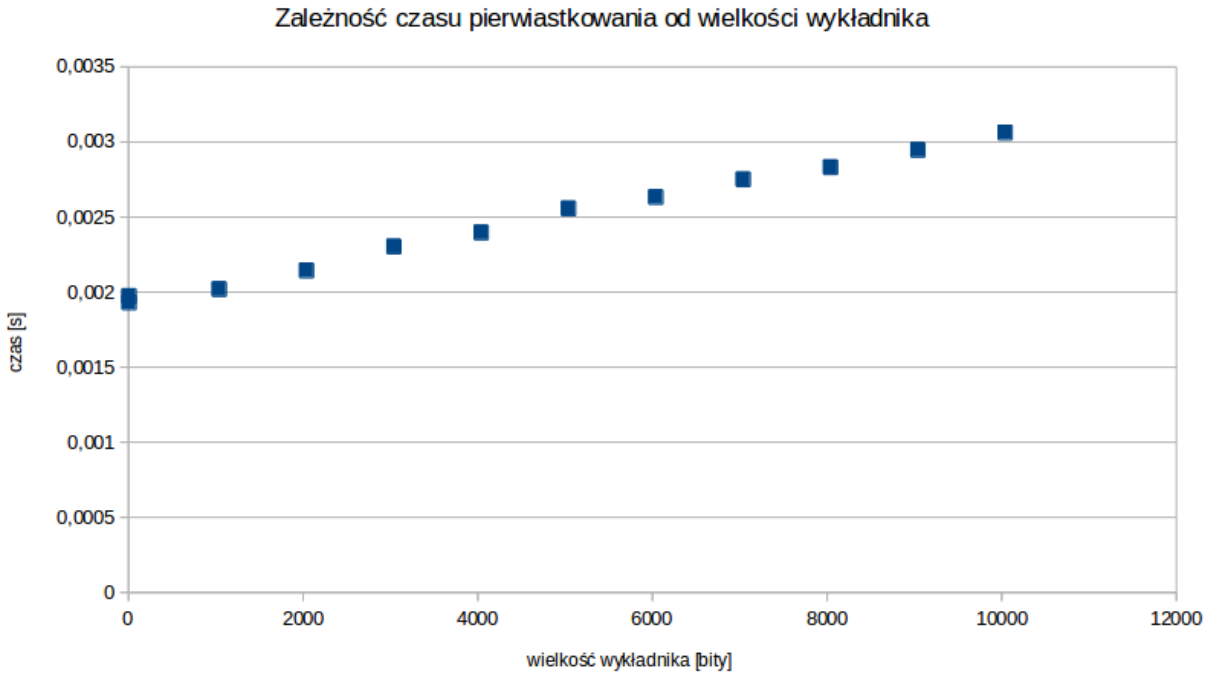

Zależność czasu mnożenia od wielkości mantysy
0,0007
0,0006
0,0005
0,0004
0,0003
0,0002
0,0001
0
czas [s]
0
100
200
300
400
500
600
długość mantysy [bity]


Zależność czasu mnożenia od wielkości wykładnika
0,001
0,0009
0,0008
0,0007
0,0006
0,0005
0,0004
0,0003
0,0002
0,0001
0
czas [s]
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
wielkość wykładnika [bity]

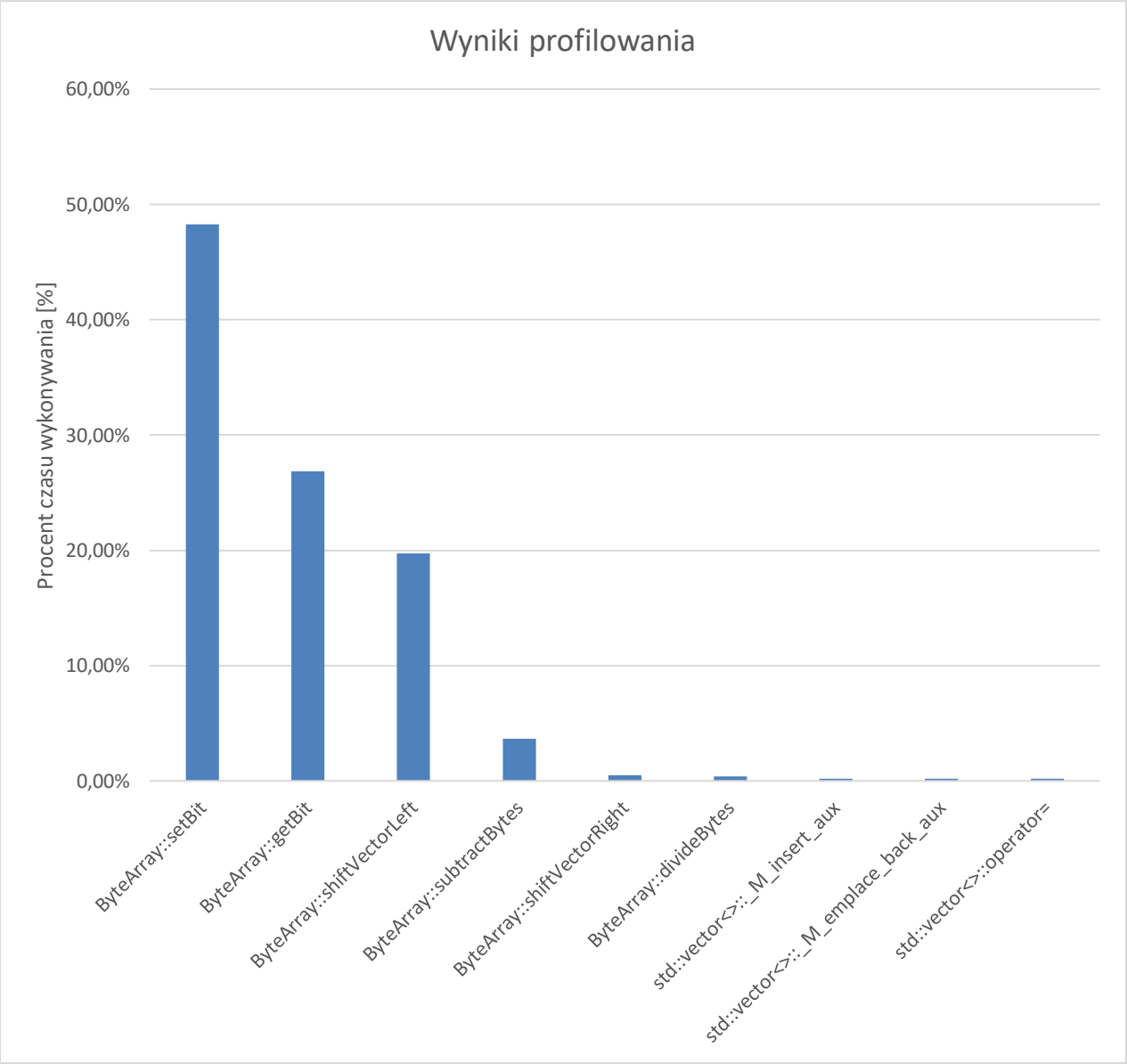

Zależność czasu dzielenia od wielości mantysy
czas [s]
0,5
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
0
20
40
60
80
100
120
140
160
wielkość mantysy [bity]


Zależność czasu dzielenia od wielkości wykładnika
czas [s]
0,001
0,0009
0,0008
0,0007
0,0006
0,0005
0,0004
0,0003
0,0002
0,0001
0
0
2000
4000
6000
8000
10000
12000
wielkość wykładnika [bity]


Zależność czasu pierwiastkowania od wielkości mantysy
0,012
0,01
0,008
0,006
0,004
0,002
0
czas [s]
0
100
200
300
400
500
600
wielkość mantysy [bity]


Zależność czasu pierwiastkowania od wielkości wykładnika
0,0035
0,003
0,0025
0,002
0,0015
0,001
0,0005
0
czas [s]
0
2000
4000
6000
8000
10000
12000
wielkość wykładnika [bity]

3.2. Profilowanie – wyniki



Rysunek 2 - wyniki profilowania

4. Wnioski

Arytmetyka liczb zmiennoprzecinkowych, z poprawnym zaokrąglaniem wielkości jest trudna do zaimplementowania. Zrealizowana przez nas biblioteka cechuje się dowolną dokładnością obliczeń, jednak złożoność obliczeniowa operacji jest wysoka. Przykładowymi zastosowaniami biblioteki mogą być:

- Podstawa implementacji bibliotek analizy numerycznej.
- Generowanie fraktali.
- Przeprowadzanie obliczeń macierzowych dla liczb zespolonych.

Pliki źródłowe projektu dostępne są pod adresem: https://github.com/c1rcle/oiakfp. Wersja elektroniczna dokumentacji znajduje się również w tym repozytorium (folder docs).

5. Bibliografia

[1] Materiały dla studentów – http://www.zak.ict.pwr.wroc.pl/materials/architektura/

[2] IEEE 754 – https://en.wikipedia.org/wiki/IEEE_754

[3] Janusz Biernat – Architektura Komputerów

[4] Division algorithm – restoring division – https://en.wikipedia.org/wiki/Division_algorithm#Restoring_division

[5] Tadeusz Tomczak – Arytmetyka Komputerów – prezentacje

[6] GNU C Library master sources - https://sourceware.org/git/?p=glibc.git