Algorytm Molera-Morissona

Aleksander Czeszejko-Sochacki

12 listopada 2017

1 Wprowadzenie

Algorytm Molera-Morissona to iteracyjna metoda obliczania wyrażeń postaci $\sqrt{a^2 + b^2}$[1]. Tego typu funkcje pojawiają się nadzwyczaj często, między innymi przy obliczaniu:

- przeciwprostokątnej
- modułu liczby zespolonej
- normy wektora
- zmiany układu współrzędnych z eukidesowych na biegunowe

1.1 Pseudokod

Algorithm 1 Algorytm iteracyjny Molera-Morissona

$p := max(|a|, |b|)$
$q := min(|a|, |b|)$
while wartość q jest znacząca **do**
  $r := (\frac{q}{p})^2$
  $s := \frac{r}{4+r}$
  $p := p + 2 * s * p$
  $q := s * q$
end while
return p

Od razu możemy zauważyć zalety algorytmu - nie wymaga on używania skomplikowanych operacji arytmetycznych, a każda iteracja składa się z kilku elementarnych działań i przypisów.

[1] a, b $\in \mathbb{R}$, najlepiej gdy a, b $\in \mathbb{Q}$, ponieważ przybliżona reprezentacja liczb niewymiernych może powiększać ewentualny błąd

1.2 Liczba iteracji

Dowodzi się, że po 3 iteracjach algorytmu błąd względny jest na poziomie $0.5*10^{-20}$, co uznamy za wystarczającą dokładność w naszym artykule. W przypadku chęci szerszej analizy zbieżności i numerycznej poprawności odsyłamy Czytelnika do rozdziału czwartego `https://blogs.mathworks.com/images/cleve/moler_morrison.pdf`

1.3 Opis użytych metod

W swoim sprawozdaniu zaprezentuję obliczenia wykonane w języku Julia. Przeprowadzone zostały w dwóch arytmetykach:

1. double, tzn. Float64
2. BigFloat(128)

Dlaczego takie arytmetyki? Precyzja arytmetyki Float64 może okazać się niewystarczająca, ponieważ wynosi ona

$$\frac{1}{2}2^{-52} = \frac{1}{8}(2^{-10})^5 \approx 10^{-16}$$

Numer iteracji	Błąd względny dla pary (3, 4)	Błąd względny dla pary (-5, 12)
2.	5.162349481224737e-9,	5.241619104568739e-13
3.	1.7763568394002506e-16,	1.3664283380001927e-16

Powyższe wyniki zgadzają się z naszym wnioskiem. Powyższe błędy po trzeciej iteracji powinny być rzędu najwyżej 10^{-21}. Powinniśmy dobrać taką arytmetykę, której precyzja będzie mniejsza niż $\frac{1}{2}2^{-20}$. Zakładając, że zwiększając w Julii liczbę bitów przydzielanych dla arytmetyki dwukrotnie, w przybliżeniu dwukrotnie zwiększa się liczba bitów przydzielanych na mantysę, precyzja arytmetyki BigFloat(128) wynosi około:

$$\frac{1}{2}2^{-100} = \frac{1}{2}(2^{-10})^{10} \approx 10^{-31}$$

co zdecydowanie wystarczy na wychwycenie ewentualnego błędu algorytmu.

Za wartość dokładną $\sqrt{a}$ będę uznawał wartość funkcji sqrt(a) w arytmetyce BigFloat(128)

2 Analiza Algorytmu

2.1 Testownie algorytmu dla wybranych danych

Poniżej przeprowadzimy testy dla następujących danych:

$$(3,4), (-5,12), (7,-24), (1,1), (1000000000,2), (71075075103, 1000000000)$$

Testy (trzy iteracje:)

(3, 4), dokładna wartość: 5.000000000000000000000000000000000000000
Małe wartości obu danych, całkowity wynik

Numer iteracji	Wynik algorytmu (Float64)	Wynik algorytmu (BigFloat(128))
1	4.986301369863014	4.986301369863013698630136986301369863002
2	4.999999974188253	4.999999741882521494926610618865305755 95
3	5.000000000000001	4.999999999999999999999999828030176654495

(-5, 12), dokładna wartość: 1.300000000000000000000000000000000000000e+01
Pierwsza dana ujemna, całkowity wynik

Numer iteracji	Wynik algorytmu (Float64)	Wynik algorytmu (BigFloat(128))
1	12.998336106489186	1.29983361064891846921797004991680532445 7e+01
2	12.999999999993186	1.29999999999931842560000017867063951355e+01
3	13.000000000000002	1.29999999999999999999999999999999999948e+01

(7, -24), dokładna wartość: 2.500000000000000000000000000000000000000e+01
Druga dana ujemna, całkowity wynik

Numer iteracji	Wynik algorytmu (Float64)	Wynik algorytmu (BigFloat(128))
1	24.999575010624735	2.49995750106247343816404589885252868678 3e+01
2	24.99999999999997	2.49999999999999692952699092177388096033 7e+01
3	25.000000000000004	2.50000000000000000000000000000000000000 0e+01

(1, 1), dokładna wartość: 1.41421356237309504880168872420969807856 9
Równe obie dane, nie dają całkowitego wyniku

Numer iteracji	Wynik algorytmu (Float64)	Wynik algorytmu (BigFloat(128))
1	1.4	1.39999999999999999999999999999999999999
2	1.4142131979695431	1.41421319796954314720812182741116751268 7
3	1.4142135623730951	1.41421356237309504879564008075425994635

(1000000000, 2), dokładna wartość: 1.000000000000000001999999999999999998001e+09
Duża pierwsza dana, duża względna różnica, nie dają całkowitego wyniku

Numer iteracji	Wynik algorytmu (Float64)	Wynik algorytmu (BigFloat(128))
1	1.0e9	1.000000000000000001999999999999999998001e+09
2	1.0e9	1.000000000000000001999999999999999998001e+09
3	1.0e9	1.000000000000000001999999999999999998001e+09

(71075075103, 1000000000), dokładna wartość: 7.10821095698284017987183842809050504809 2e+10
Duże obie dane, nie dają całkowitego wyniku

Numer iteracji	Wynik algorytmu (Float64)	Wynik algorytmu (BigFloat(128))
1	7.108210956981117e10	7.10821095698111760772695745828990150852 3e+10
2	7.10821095698284e10	7.10821095698284017987183842809050504809 2e+10
3	7.10821095698284e10	7.10821095698284017987183842809050504813 2e+10

Możemy w powyższych przykładach zaobserwować następujące zjawiska:

- Zmniejszenie precyzji arytmetyki powoduje zwiększenie liczby cyfr znaczących
- Algorytm nie wyliczył oczekiwanych wartości całkowitych dla pierwszych trzech par

Więcej obserwacji będziemy w stanie wyciągnąć po analizie błędów względnych.

Błędy względne [2]

Dane	$\frac{\lvert\Delta y\rvert}{y}$, $\tilde{y}$ we Float64	$\frac{\lvert\Delta y\rvert}{y}$, $\tilde{y}$ w BigFloat(128)
(3, 4)	1.776356839400250464677810668945312500001e-16	0.0
(-5, 12)	1.366428338000192665136777437650240384618e-16	0.0
(7, -24)	1.421085471520200371742248535156250000001e-16	0.0
(1, 1)	0.000000000000000000000000000000000000000	0.0
(1000000000, 2)	0.000000000000000000000000000000000000000	0.0
(71075075103, 1000000000)	0.000000000000000000000000000000000000000	0.0

[2] dla przejrzystości zmniejszyłem liczbę nieznaczących zer w błędach w arytmetyce BigFloat(128); $\tilde{y}$ - wynik algorytmu, y - wartość obliczona przy pomocy funkcji sqrt()

Obserwacje (w powyższych przykładach):

- Błędy w arytmetyce Float64 są najwyżej rzędu 10^{-16}, co zgadza się z przybliżoną wartością precyzji tej arytmetyki obliczoną we wprowadzeniu
- W niektórych przypadkach błędy są zerowe (lub mniejsze niż precyzja arytmetyki)
- W arytemtyce BigFloat(128) wszyskie błędy są zerowe (lub mniejsze niż precyzja arytmetyki)
- Błedy w arytmetyce BigFloat(128) są niewiększe niż we Float64

Hipoteza 1 *Dla dostatecznie dużej precyzji algorytm zwraca matematycznie dokładny wynik dla dowolnych danych (nie wiadomo, czy taka precyzja jest osiągalna na każdej maszynie).*

2.2 Ciekawostka

Algorytm zachowuje się ciekawie, gdy opuścimy wartości bezwzględne w inicjalizacji zmiennych. Po trzeciej iteracji

$$\sqrt{(-7)^2 + 24^2} \neq \sqrt{7^2 + (-24)^2}$$

natomiat po czwartej

$$\sqrt{(-7)^2 + 24^2} = \sqrt{7^2 + (-24)^2}$$

Zachęcam Czytelnika do przeprowadzenia własnych eksperymentów.

2.3 Zalety stosowania algorytmu

Algorytm iteracyjny Molera-Morissona ma bardzo przyzwoity wykładnik zbieżności - z każdą iteracją liczba cyfr znaczących zwiększa się około trzykrotnie. Niewątpliwą jego przewagą nad liczeniem wyrażenia przy pomocy funkcji sqrt() jest fakt, że w algorytmie nie podnosimy danych do kwadratu, przez co ma dużo mniejszą złożoność pamięciową. Złożoność czasowa zależy od czasu wykonywania elementarnych operacji (chyba że chcemy wykonać więcej niż trzy iteracje).

3 Zasosowanie algorytmu Molera-Morissona do obliczania normy wektora

Normą wektora x nazywamy wyrażenie:

$$\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

Możemy ją jednak zapisać następująco:

$$\begin{aligned}\sqrt{\sum_{i=1}^{n} x_i^2} &= \underbrace{\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}}_{n} \\ &= \sqrt{\sqrt{(\sqrt{x_1^2 + x_2^2})^2 + x_3^2}^{\,2} + \cdots + x_n^2} \\ &= \sqrt{\sqrt{\sqrt{\sqrt{(\sqrt{x_1^2 + x_2^2})^2 + x_3^2}^{\,2} + \cdots + x_{n-1}}^{\,2} + x_n^2}}\end{aligned} \tag{1}$$

3.1 Pseudokod obliczania normy wektora

Powyższe rozpisanie umożliwia nam zapisanie algorytmu:

Algorithm 2 Algorytm iteracyjny Molera-Morissona

```
result = x_1
for i in 2, 3, ..., n do
    result = Moler-Morrison(result, x_i)
end for
return result
```

Dokładność otrzymanego wyniku będzie zależna od dokładności samego algorytmu, jak i kolejności sumowania, jednak problem kolejności sumowania dla uzyskania najmniejszego błędu to temat obszerny i nie będzie on rozważany w tym artykule.

Literatura

[1] C. Moler, D. Morrison: Replacing Square Roots by Pythagorean Sums