Obliczenia Naukowe Lista 2 Laboratoria

Piotr Szyma

1 listopada 2017

1 Zadanie 1

1.1 Opis problemu

Powtórz zadanie 5 z listy 1, ale usuń ostatnią 9 z x_4 i ostatnią 7 z x_5. Jaki wpływ na wyniki mają niewielkie zmiany danych?

1.2 Rozwiązanie

Uruchomiłem program z pierwszej listy po dokonaniu modyfikacji wejścia. Wyniki zestawiłem w tabeli poniżej. Numeracja algorytmów to odpowiednio 1, 2 - algorytmy sumowania odpowiednio w górę i w dół, 3, 4 - sumy częściowe. Kody źródłowe zostały dołączone do sprawozdania.

1.3 Wynik

W przypadku arytmetyki Float64 lekka zmiana wektorów przyczyniła się do zmiany ostatecznego wyniku - wciąż znacznie odbiegajacego od realnych wartości. Dla arytmetyki Float32 usunięcie ostatnich cyfr nie miało wpływu na ostateczny wynik, gdyż arytmetyka nie jest wystarczająco dokładna.

Liczby pojedynczej precyzji pozwalają na zapisanie do 7 cyfr znaczących w reprezentacji dziesiętnej. Zmiana w zadaniu dotyczyła cyfr na pozycji 10, dlatego nie doprowadziło to do zmiany.

Float64 natomiast daje większą dokładność, dlatego zmiana miała wpływ na wynik. Zmniejszenie dokładności składowych wektora pozwoliło na bliższe sobie wyniki obliczeń każdego z algorytmów do 9 miejsca po przecinku. Od 10 cyfry natomiast wyniki zaczęły się rozbiegać.

Poniżej przedstawiłem tabelę z wynikami obliczeń.

Float64

$algorytm$	x_{przed}	x_{po}	δx
1	$1.0251881368e-10$	$-4.2963427399e-03$	$4.2963428424e-03$
2	$-1.5643308870e-10$	$-4.2963429987e-03$	$4.2963428423e-03$
3	$0.0000000000e+00$	$-4.2963428423e-03$	$4.2963428423e-03$
4	$0.0000000000e+00$	$-4.2963428423e-03$	$4.2963428423e-03$

Float32

$algorytm$	x_{przed}	x_{po}	δx
1	$-4.9994429946e-01$	$-4.9994429946e-01$	$0.0000000000e+00$
2	$-4.5434570313e-01$	$-4.5434570313e-01$	$0.0000000000e+00$
3	$-5.0000000000e-01$	$-5.0000000000e-01$	$0.0000000000e+00$
4	$-5.0000000000e-01$	$-5.0000000000e-01$	$0.0000000000e+00$

2 Zadanie 2

2.1 Opis problemu

Narysować wykres funkcji $f(x) = e^x \ln(1+e^{-x})$ w co najmniej dwóch dowolnych programach do wizualizacji. Następnie policzyć granicę funkcji $\lim_{x\to\infty} f(x)$. Porównać wykres funkcji z policzoną granicą. Wyjaśnić zjawisko.

2.2 Rozwiązanie

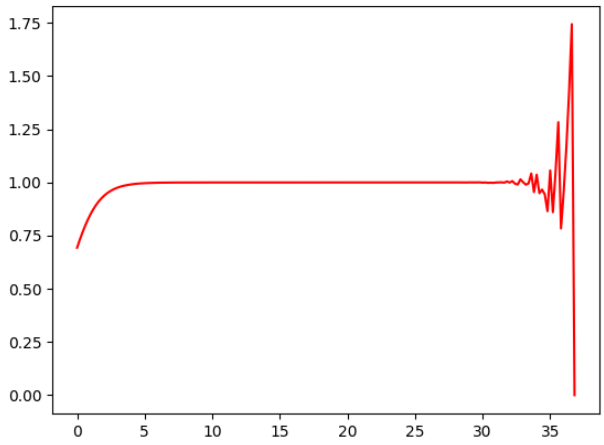
Oto granica funkcji $f(x)$:

$$\lim_{x\to\infty} e^x \ln(1+e^{-x}) = 1$$

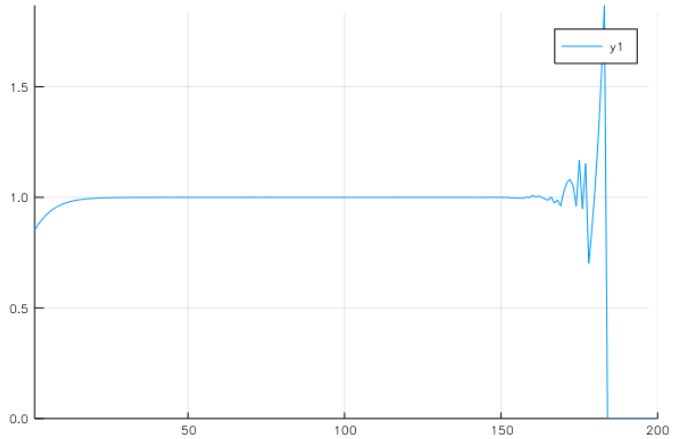
2.3 Wynik

Wykresy wygenerowane za pomocą dwóch programów do wizualzacji załączone są na Rysunkach 1 i 2. Kody źródłowe programów dołączyłem do sprawozdania w osobnych plikach.

Jak widać na wykresach, w pewnym momencie funkcja zaczyna przyjmować nieoczekiwane - błędne - wartości. Głównym czynnikiem wpływającym na takie zachowanie jest operacja mnożenia bardzo dużej liczby e^x przez bardzo małą liczbę $\ln(1+e^{-x})$, bo $\lim_{x\to\infty} \ln(1+e^{-x}) = 0$.



Rysunek 1: Wykres wygenerowany za pomocą biblioteki matplotlib w języku Python



Rysunek 2: Wykres wygenerowany za pomocą biblioteki Plot w języku Julia

3 Zadanie 3

3.1 Opis problemu

Rozwiązać układy równań podane w specyfikacji zadania za pomocą dwóch algorytmów: eliminacji Gaussa ($\texttt{x} = \texttt{A/b}$) oraz $x = A^{-1}b$ ($\texttt{x} = \texttt{inv(A)} * \texttt{b}$). Eksperymenty wykonać dla macierzy Hilberta $\mathbf{H_n}$ z rosnącym stopniem n > 1 oraz dla macierzy losowej R_n, n = 5, 10, 20 z rosnącym wskaźnikiem uwarunkowania $c = 10, 10^3, 10^7, 10^{12}, 10^{16}$. Porównać obliczony $\tilde{x}$ z rozwiązaniem dokładnym $x = (1, ..., 1)T$, tj. policzyć błędy względne.

W języku Julia za pomocą funkcji `cond(A)` można sprawdzić jaki jest wskaźnik uwarunkowania wygenerowanej macierzy. Natomiast za pomocą funkcji `rank(A)` można sprawdzić jaki jest rząd macierzy.

3.2 Rozwiązanie

W celu wykonania podanych w treści zadania obliczeń, stworzyłem w języku Julia odpowiednie funkcje. Kody źródłowe programów dołączyłem do sprawozdania. Błędy względne obliczeń wyliczyłem przy pomocy normy wektora wg. wzoru:

$$\delta = \frac{||\widetilde{x} - x||}{||x||}$$

3.3 Wynik

W przypadku macierzy losowych zaobserwowałem, że błąd bezwzględny, niezależnie od stopnia macierzy, oscyluje w granicach 10^{-16}. Zupełnie inaczej wyglądało to w przypadku macierzy Hilberta.

Macierz Hilberta jest postaci $h_{ij} = \frac{1}{i+j-1}$. Jak widać, wiele z komórek macierzy nie da się zapisać w postaci skończonej w reprezentacji biarnej, np. $\frac{1}{3}, \frac{1}{6}$. Macierz Hilberta jest macierzą źle uwarunkowaną - wykonując obliczenia z jej użyciem jesteśmy narażeni na duży wpływ błędu wynikajacego z reprezentacji numerycznej, dlatego rozwiązywanie w numeryczny sposób nawet małych układów równań z wykorzystaniem macierzy Hilberta jest niemożliwe.

Poniżej przedstawiłem tabelę - zestawienie wyników w postaci wyliczonych błędów względnych.

Macierz Hilberta

n	$cond$	$gauss$	A^{-1}
1	$1.00000e+00$	$0.0000000000e+00$	$0.0000000000e+00$
2	$1.92815e+01$	$5.6610488670e-16$	$1.1240151438e-15$
3	$5.24057e+02$	$8.0225937723e-15$	$9.8255260382e-15$
4	$1.55137e+04$	$4.4515459602e-13$	$2.9504776373e-13$
5	$4.76607e+05$	$1.6828426299e-12$	$8.5000557778e-12$
6	$1.49511e+07$	$2.6189133023e-10$	$3.3474135070e-10$
7	$4.75367e+08$	$1.2606867224e-08$	$5.1639591836e-09$
8	$1.52576e+10$	$1.0265430657e-07$	$2.6987150743e-07$
9	$4.93154e+11$	$4.8323571205e-06$	$9.1758468686e-06$
10	$1.60244e+13$	$6.3291537230e-04$	$4.5521422517e-04$
11	$5.22268e+14$	$1.1543958596e-02$	$8.0444667734e-03$
12	$1.75147e+16$	$2.9756403107e-01$	$3.4392937091e-01$
13	$3.34414e+18$	$2.3750178677e+00$	$5.5857968932e+00$
14	$6.20079e+17$	$5.2810046468e+00$	$4.8006419290e+00$
15	$3.67439e+17$	$1.1772947348e+00$	$4.8273577213e+00$
16	$7.86547e+17$	$2.0564655824e+01$	$3.1736467496e+01$
17	$1.26368e+18$	$1.7742214635e+01$	$1.5910335963e+01$
18	$2.24463e+18$	$4.2764564411e+00$	$6.2812234335e+00$
19	$6.47195e+18$	$2.2119937293e+01$	$2.2925614016e+01$
20	$1.35537e+18$	$1.4930069669e+01$	$2.1539498603e+01$

Losowa macierz o zadanym współczynniku *cond*

n	$cond$	$gauss$	A^{-1}
5	1	$3.4399002280e-16$	$3.3306690739e-16$
5	3	$1.7199501140e-16$	$1.9860273226e-16$
5	7	$1.7199501140e-16$	$2.6272671963e-16$
5	12	$2.6272671963e-16$	$1.1102230246e-16$
5	16	$2.8086667749e-16$	$3.2934537262e-16$
10	1	$3.4755478145e-16$	$1.7901808365e-16$
10	3	$1.8906416839e-16$	$2.2480302876e-16$
10	7	$2.2204460493e-16$	$3.9094986940e-16$
10	12	$2.9790409839e-16$	$7.1521115290e-16$
10	16	$3.2558130189e-16$	$6.4258803294e-16$
20	1	$7.3685695350e-16$	$3.9409007944e-16$
20	3	$4.8963226964e-16$	$4.3990617274e-16$
20	7	$4.7298628134e-16$	$6.2063353831e-16$
20	12	$4.4478255798e-16$	$4.5097472449e-16$
20	16	$4.9214322083e-16$	$4.5301997183e-16$

4 Zadanie 4

4.1 Opis problemu

„Złośliwy wielomian" Wilkinsona. Zainstalować pakiet `Polynomials`.

(a) Użyć funkcji `roots` z pakietu `Polynomials` do obliczenia 20 zer wielomianu P zadanego w treści zadania. Sprawdzić obliczone pierwiastki $z_k, 1 \leq k \leq 20$, obliczając $|P(z_k)|$, $|p(z_k)|$ i $|z_k - k|$. Wyjaśnić rozbieżność. Zapoznać się z funkcjami `Poly`, `poly`, `polyval` z pakietu `Polynomials`.

(b) Powtórzyć eksperyment Wilkinsona, tj. zmienić współczynnik -210 na $-210 - 2^{-23}$. Wyjaśnić zjawisko.

4.2 Rozwiązanie

Funkcja `Poly(x)` generuje wielomian w zależności od współczynników podanych jako parametr. Funkcja `poly(x)` tworzy wielomian na podstawie pierwiastków wielomianu, natomiast funkcja `polyval(p, x)` pozwala na wyliczenie wartości wielomianu `p` dla zadanego argumentu `x`. Napisałem program z użyciem wyżej wymienionych funkcji z biblioteki `Polynomials`. Kody źródłowe dołaczyłem do sprawozdania.

4.3 Wynik

Wyniki poszczególnych części zadań zamieściłem w tabelach poniżej - zatytułowanych odpowiednio (a) i (b).

Rozwiązując podpunkt (a) zadania zaobserwowałem pewne rozbieżności w wynikach. Porównując wartości $P(z_k)$ oraz $p(z_k)$ zauważyłem, że wyniki różnią się od siebie. Wyliczone za pomocą funkcji `roots(P(x))` pierwiastki wielomianu P różnią się od jego rzeczywistych pierwiastków. Wnioskiem z poczynionych obserwacji jest fakt, że operacje wyznaczania wielomianu na podstawie współczynników czy tworzenia go z pierwiastków, są obarczone pewnym zaburzeniem, wynikającym z precyzji arytmetyki użytej do wykonania obliczeń.

Podpunkt (b) dotyczył złośliwego wielomianu Wilkinsona - zaburzania wielomianów tak, aby wykonywane na nich operacje numeryczne generowały duże rozbieżności od rzeczywistych wyników. Mała zmiana jednego ze współczynników sprawiła, że wyniki w podpunkcie (b) znacznie różniły się od punktu (a). Ponad to, funkcja `roots(P'(x))` wywołana na wielomianie z drugiego przykładu, zwróciła pierwiastki zespolone.

Wyniki (a)

k	$\lvert P(z_k)\rvert$	$\lvert p(z_k)\rvert$	$\lvert z_k - k\rvert$
1	36352.0	38400.0	$3.0109248427834245e-13$
2	181760.0	198144.0	$2.8318236644508943e-11$
3	209408.0	301568.0	$4.0790348876384996e-10$
4	$3.106816e6$	$2.844672e6$	$1.626246826091915e-8$
5	$2.4114688e7$	$2.3346688e7$	$6.657697912970661e-7$
6	$1.20152064e8$	$1.1882496e8$	$1.0754175226779239e-5$
7	$4.80398336e8$	$4.78290944e8$	0.00010200279300764947
8	$1.682691072e9$	$1.67849728e9$	0.0006441703922384079
9	$4.465326592e9$	$4.457859584e9$	0.002915294362052734
10	$1.2707126784e10$	$1.2696907264e10$	0.009586957518274986
11	$3.5759895552e10$	$3.5743469056e10$	0.025022932909317674
12	$7.216771584e10$	$7.2146650624e10$	0.04671674615314281
13	$2.15723629056e11$	$2.15696330752e11$	0.07431403244734014
14	$3.65383250944e11$	$3.653447936e11$	0.08524440819787316
15	$6.13987753472e11$	$6.13938415616e11$	0.07549379969947623
16	$1.555027751936e12$	$1.554961097216e12$	0.05371328339202819
17	$3.777623778304e12$	$3.777532946944e12$	0.025427146237412046
18	$7.199554861056e12$	$7.1994474752e12$	0.009078647283519814
19	$1.0278376162816e13$	$1.0278235656704e13$	0.0019098182994383706
20	$2.7462952745472e13$	$2.7462788907008e13$	0.00019070876336257925

Wyniki (b)

k	$\lvert P(z_k)\rvert$	$\lvert z_k - k\rvert$
1	20992.0	$1.6431300764452317e-13$
2	349184.0	$5.503730804434781e-11$
3	$2.221568e6$	$3.3965799062229962e-9$
4	$1.046784e7$	$8.972436216225788e-8$
5	$3.9463936e7$	$1.4261120897529622e-6$
6	$1.29148416e8$	$2.0476673030955794e-5$
7	$3.88123136e8$	0.00039792957757978087
8	$1.072547328e9$	0.007772029099445632
9	$3.065575424e9$	0.0841836320674414
10	$7.143113638035824e9$	0.6519586830380406
11	$7.143113638035824e9$	1.1109180272716561
12	$3.357756113171857e10$	1.665281290598479
13	$3.357756113171857e10$	2.045820276678428
14	$1.0612064533081976e11$	2.5188358711909045
15	$1.0612064533081976e11$	2.7128805312847097
16	$3.315103475981763e11$	2.9060018735375106
17	$3.315103475981763e11$	2.825483521349608
18	$9.539424609817828e12$	2.454021446312976
19	$9.539424609817828e12$	2.004329444309949
20	$1.114453504512e13$	0.8469102151947894

Wartość zwracana przez funkcję `roots`() dla wielomianu z podpunktu (b)
ComplexFloat64[1.0+0.0im, 2.0+0.0im, 3.0+0.0im, 4.0+0.0im, 5.0+0.0im, 6.00002+0.0im, 6.9996+0.0im, 8.00777+0.0im, 8.91582+0.0im, 10.0955-0.644933im, 10.0955+0.644933im, 11.7939-1.65248im, 11.7939+1.65248im, 13.9924-2.51882im, 13.9924+2.51882im, 16.7307-2.81262im, 16.7307+2.81262im, 19.5024-1.94033im, 19.5024+1.94033im, 20.8469+0.0im]

5 Zadanie 5

5.1 Opis problemu

Równanie rekurencyjne (model logistyczny, model wzrostu populacji)

$$p_{n+1} := p_n + rp_n(1 - p_n)\ dla\ n = 0, 1, 2, \ldots$$

Zadanie polegało na przeprowadzeniu następującego eksperymentu:

(a) Dla danych $p_0 = 0.01$ oraz $r = 3$ wykonać 40 iteracji w/w wyrażenia (1), następnie 40 iteracji z modyfikacją - po 10 iteracji obciąć wynik (2), odrzucając cyfry po 3 miejscu po przecinku. Obliczenia wykonać w arytmetyce `Float32`. Porównać wyniki.

(b) Porównać wyliczanie 40 iteracji dla arytmetyk `Float32` oraz `Float64`.

5.2 Rozwiązanie

W celu wyliczenia zadanych rekurencji stworzyłem program w Julii. Kody źródłowe dołączyłem w osobnym pliku.

5.3 Wynik

Wyniki obliczeń przedstawiłem w tabeli poniżej.

Pierwszym ważnym spostrzeżeniem jest fakt, że z każdą kolejną iteracją, ilość cyfr po przecinku rośnie dwukrotnie. Nieubłagalnie zbliżając się już w drugiej iteracji do granicy dokładności arytmetyki. W trzeciej iteracji liczba jest już przybliżana - pojawiają się pierwsze rozbieżności. Każda kolejna iteracja to kolejny błąd, który coraz mocniej wpływa na wynik.

W punkcie (a) do 10 iteracji wyniki nie różniły się od siebie. Po obcięciu cyfr znaczących w przykładzie (2), kolejne iteracje zaczęły coraz bardziej się różnić. W 40 iteracji wynik po obcięciu różnił się od pierwotnej wersji 4 krotnie. Ta część zadania pokazuje, jak ważna jest dokładność obliczeń. Ucięcie liczby w kolejnych iteracjach doprowadza do coraz większych różnic wyniku.

Podpunkt (b) to porównanie arytmetyk `Float32` oraz `Float64`. Tutaj również zawodzi skończona precyzja arytmetyk. Pierwsze iteracje - w ramach dostępnych dla arytmetyk precyzji, dają poprawne wyniki. Z każdą kolejną jednak, gdy ilość miejsc po przecinku rośnie, arytmetyki ucinają cyfry i przybliżają - generując błąd. Patrząc na pierwsze iteracje - wyniki dla obydwu arytmetyk wykazują jakieś podobieństwo. 20 iteracja to ostatnia, w której widoczne jest jakiekolwiek podobieństwo. (cyfra 5 będąca pierwszą po przecinku)

Z zadania płyną następujące wnioski: niezależnie od tego, jaką arytmetykę komputera (skończoną) obierzemy - numeryczne obliczenia prędzej czy później doprowadzą do utraty przewidywalności.

Wyniki (a)

n	(1)	(2)
1	0.039700001478195	0.039700001478195
2	0.154071733355522	0.154071733355522
3	0.545072615146637	0.545072615146637
4	1.288978099822998	1.288978099822998
5	0.171518802642822	0.171518802642822
10	0.722930610179901	0.722000002861023
15	1.270483732223511	1.257216930389404
20	0.579903602600098	1.309691071510315
25	1.007080554962158	1.092910766601563
30	0.752920925617218	1.319182157516479
35	1.021098971366882	0.034241437911987
40	0.258605480194092	1.093567967414856

Wyniki (b)

n	$Float32$	$Float64$
1	0.039700001478195	0.039700000000000
2	0.154071733355522	0.154071730000000
3	0.545072615146637	0.545072626044421
4	1.288978099822998	1.288978001188801
5	0.171518802642822	0.171519142109176
10	0.722930610179901	0.722914301179573
15	1.270483732223511	1.270261773935077
20	0.579903602600098	0.596529312494691
25	1.007080554962158	1.315588346001072
30	0.752920925617218	0.374146489639287
35	1.021098971366882	0.925382128557105
40	0.258605480194092	0.011611238029749

6 Zadanie 6

6.1 Opis problemu

Dla zadanego równania rekurencyjnego przeprowadź serię eksperymentów.

$$x_{n+1} = x_n^2 + c \; dla \; n = 0, 1, 2, \ldots$$

Wykonać w arytmetyce `Float64` 40 iteracji wyrażenia dla 7 zestawów danych:

1. $c = -2 \; i \; x_0 = 1$
2. $c = -2 \; i \; x_0 = 2$
3. $c = -2 \; i \; x_0 = 1.99999999999999$
4. $c = -1 \; i \; x_0 = 1$
5. $c = -1 \; i \; x_0 = -1$
6. $c = -1 \; i \; x_0 = 0.75$
7. $c = -1 \; i \; x_0 = 0.25$

6.2 Rozwiązanie

W celu wyliczenia zadanych rekurencji stworzyłem program w Julii, którego kod źródłowy dołączyłem do sprawozdania. Wyniki obliczeń przedstawiłem w tabeli poniżej.

6.3 Wynik

Spośród wszystkich zestawów danych możemy wybrać te stabilne oraz te niestabilne.

W przypadku pierwszego zestawu danych, tj. $x_0 = 1$ i $c = -2$, wyniki obliczeń pokrywają się z przewidywaniami. Każda iteracja zwraca -1. Drugi zestaw również nie przynosi zaskoczenia - każdy wynik wynosi -2. W trzecim zestawie danych operujemy na liczbie o 14 cyfrach po przecinku - jej mnożenie generuje liczbę, która nie mieści się już w arytmetyce `Float64` i dochodzi do przybliżenia - pojawia się błąd. Każde kolejne mnożenie powoduje kolejne przybliżenie - błąd narasta. Taki algorytm z numerycznego punktu widzenia zalicza się do grupy niestabilnych i jest niepożądany.

Ciągi 4 oraz 5 tworzą cykl składający się z liczb całkowitych, w ich przypadku nigdy nie dojdzie do wyskoczenia poza precyzję arytmetyki. Te zestawy danych są stabilne.

Ostatnie dwa ciągi są o tyle interesujące, że wartość, do której - z matematycznego punktu widzenia - dążą, osiągana jest już na wysokości 13 iteracji. Wynika to z ograniczeń precyzji arytmetyki. Liczby są na tyle małe, że arytmetyka wymaga ich zaokrąglenia.

$^{algorytm}/_{iteracja}$	1	2	4	5
1	−1.000000000	2.000000000	0.000000000	0.000000000
2	−1.000000000	2.000000000	−1.000000000	−1.000000000
3	−1.000000000	2.000000000	0.000000000	0.000000000
4	−1.000000000	2.000000000	−1.000000000	−1.000000000
5	−1.000000000	2.000000000	0.000000000	0.000000000
10	−1.000000000	2.000000000	−1.000000000	−1.000000000
15	−1.000000000	2.000000000	0.000000000	0.000000000
20	−1.000000000	2.000000000	−1.000000000	−1.000000000
25	−1.000000000	2.000000000	0.000000000	0.000000000
30	−1.000000000	2.000000000	−1.000000000	−1.000000000
35	−1.000000000	2.000000000	0.000000000	0.000000000
40	−1.000000000	2.000000000	−1.000000000	−1.000000000

$^{algorytm}/_{iteracja}$	3	6	7
1	2.0000000000	−0.4375000000	−0.9375000000
2	2.0000000000	−0.8085937500	−0.1210937500
3	2.0000000000	−0.3461761475	−0.9853363037
4	2.0000000000	−0.8801620749	−0.0291123686
5	2.0000000000	−0.2253147219	−0.9991524700
10	1.9999999895	−0.9996201881	−0.0000000001
15	1.9999892712	−0.0000000000	−1.0000000000
20	1.9890237264	−1.0000000000	0.0000000000
25	−1.9550094875	0.0000000000	−1.0000000000
30	1.7385002138	−1.0000000000	0.0000000000
35	−1.3354478410	0.0000000000	−1.0000000000
40	−0.3289791230	−1.0000000000	0.0000000000