Metody Obliczeniowe w Nauce i Tehcnice
II Rok

Informatyka
Wydział Informatyki, Elektroniki i Telekomunikacji

Znajdowanie pierwiastków

laboratorium nr 6

Jacek Nitychoruk

Semestr letni
2019/2020

Spis treści

1 Porównanie działania metod obliczania pierwiastków funkcji 3
1.1 Badane funkcje . . . 3
1.2 Metoda *False Position* (*regula falsi*) . . . 6
1.3 Metoda Newtona . . . 8
1.4 Metoda Steffensena . . . 10
1.5 Podsumowanie wyników uzyskanych za pomocą trzech metod: . . . 12
1.6 Wnioski . . . 12

2 Demonstracja działania metod poszukiwania pierwiastków na wybranej funkcji 13
2.1 Badana funkcja . . . 13
2.2 Wyniki działania różnych metod obliczania pierwiastków . . . 14
2.3 Wyniki działania 12 metod obliczania pierwiastków metodą *False Position* . . . 16
2.4 Wszystkie pierwiastki funkcji . . . 17

3 Demonstracja przykładów nieprawidłowego działania 18
3.1 Metoda *False Position* . . . 18
3.2 Metoda Newtona . . . 19
3.3 Metoda Steffensena . . . 20

4 Wstęga Newtona 21
4.1 Program . . . 21
4.2 Przykładowy fraktal - wstęga Newtona . . . 22

5 Dodatek - porównanie 12 implementacji metody False Position 24

1 Porównanie działania metod obliczania pierwiastków funkcji

Zadanie: Wybrać trzy metody poszukiwania pierwiastków:

- wykorzystującą przedział i zmianę znaku,
- wykorzystującą pochodną,
- wykorzystującą przybliżenie pochodnej

Każdą z trzech wybranych metod przetestować (ilość iteracji, ilość wywołań funkcji) na sześciu wybranych funkcjach ze zbioru Zero Finder Tests. Wyniki przedstawić w formie tabelki. Pamiętać o sprawdzeniu czy wynik jest poprawny poprzez obliczenie wartości funkcji dla znalezionego pierwiastka!

1.1 Badane funkcje

Spośród 19 funkcji znajdujących się w zbiorze Zero Finder Tests wybrałem 6, które wydały mi się dość różnorodne i ciekawe.

1. $f(x) = e^x - \frac{1}{(10x)^2}$
2. $f(x) = \cos(x) - x$
3. $f(x) = \frac{20x}{100x^2+1}$
4. $f(x) = \begin{cases} -\sqrt[3]{|x|} \cdot e^{-x^2} & x < 0 \\ 0 & x = 0 \\ \sqrt[3]{|x|} \cdot e^{-x^2} & x > 0 \end{cases}$
5. $f(x) = \pi \cdot \frac{x-5}{180} - 0.8 \cdot \sin\left(\frac{x\pi}{180}\right)$
6. $f(x) = \cos(100x) - 4 \cdot \operatorname{erf}(30x - 10)$

We wzorze ostatniej badanej funkcji występuje funkcja błędu Gaussa:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, \mathrm{d}t \tag{1}$$

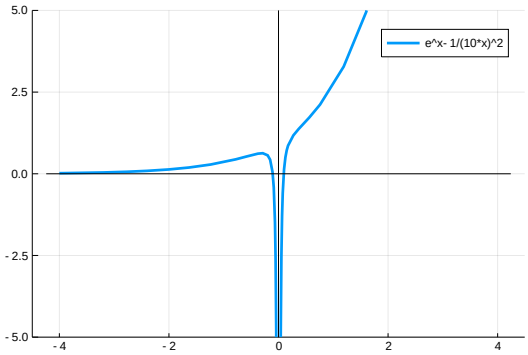
Następnie użyłem środowiska Julia, w którym wprowadziłem badane funkcje, aby móc wykonywać na nich operacje:

```
f1(x) = exp(x)-1/(10*x)^2
f2(x) = cos(x)-x
f3(x) = 20*x/(100*x^2 + 1)
function f4(x)
    if x<0
        return -(abs(x)^(1/3))*exp(-x^2)
    elseif x == 0
        return 0
    else
        return abs(x)^(1/3)*exp(-x^2)
    end
```

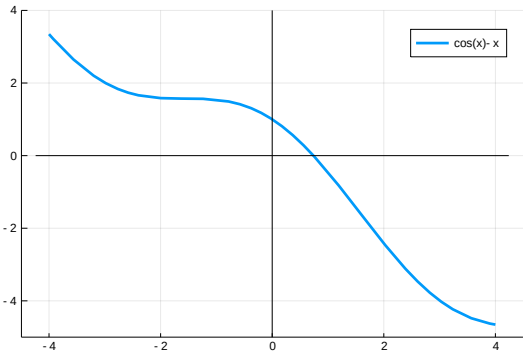
```
end
f5(x) = pi*(x-5)/180 - 0.8*sin(pi*x/180)
f6(x) = cos(100*x)-4*erf(30x-10)

D(f) = x -> ForwardDiff.derivative(f,float(x))

f = Function[]
f = [f1,f2,f3,f4,f5,f6]

x_left = Float64[];
x_left = [-4,-4,-2,-2,-5,-0.1]

x_right = Float64[];
x_right = [4,4,2,2,45,0.6]

x_search_l = Float64[];
push!(x_search_l, 0)
for i in 2:length(f)
    push!(x_search_l, x_left[i])
end

x_search_r = Float64[];
for i in 1:length(f)
    push!(x_search_r, x_right[i])
end
x_search_r[3] = 3
x_search_r[4] = 3

start = [1,1,1,0.3,1,0.32]

y_top = Float64[];
y_top = [5, 4, 1, 1, 0.2, 5.5]

y_bottom = Float64[];
y_bottom = [-5, -5, -1, -1, -0.1, -5.5]

labels = String[]
push!(labels, "e^x-1/(10*x)^2")
push!(labels, "cos(x)-x")
push!(labels, "20*x/(100*x^2+1)")
push!(labels, "+/-(|x|^(1/3))*e^(-x^2)")
push!(labels, "pi*(x-5)/180 - 0.8*sin(pi*x/180)")
push!(labels, "cos(100x)-4*erf(30x-10)")
```

Następnie wygenerowałem wykresy funkcji (rys. 1):

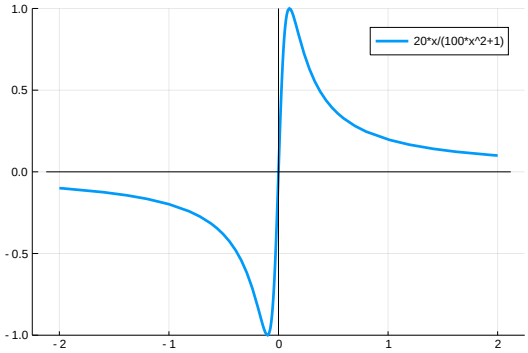
```
for i in 1:length(f)
    p = plot(f[i], x_left[i], x_right[i], ylims=(y_bottom[i],y_top[i]), label=labels[i], linewidth=3);
    plot!( x->0, label=false, color="black")
    plot!([0], seriestype = :vline, color="black", label=false)
    display(p)
    savefig(string("plot",i,".pdf"))
end
```



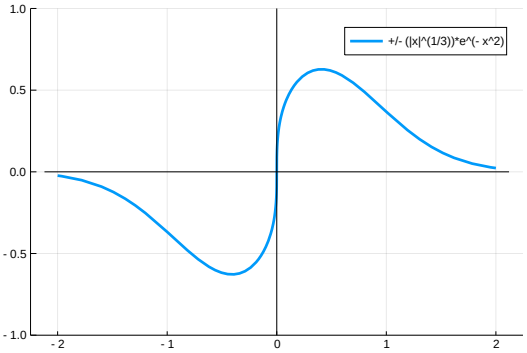
(a) $f(x) = e^x - \frac{1}{(10x)^2}$



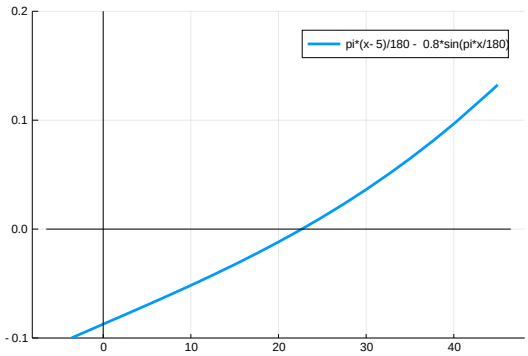
(b) $f(x) = \cos(x) - x$



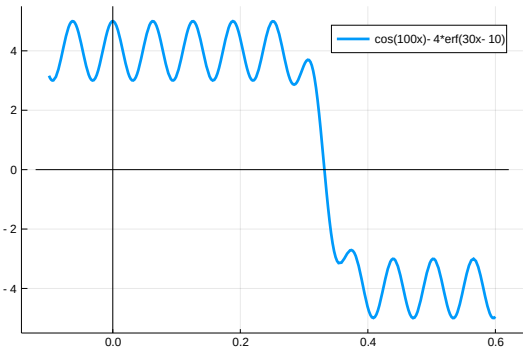
(c) $f(x) = \frac{20x}{100x^2+1}$



(d) $f(x) = \pm |x|^{\frac{1}{3}} \cdot e^{-x^2}$



(e) $f(x) = \pi \cdot \frac{x-5}{180} - 0.8 \cdot \sin\left(\frac{x\pi}{180}\right)$



(f) $f(x) = \cos(100x) - 4 \cdot \mathrm{erf}(30x - 10)$

Rysunek 1: Wykresy badanych funkcji na interesujących nas przedziałach

1.2 Metoda *False Position* (*regula falsi*)

Następnie wykonałem próby obliczenia pierwiastków przy pomocy metody wykorzystującej zmianę znaku na krańcach przedziału. W tym celu użyłem kodu poniżej:

```
for i in 1:length(f)
println(labels[i])
x = find_zero(f[i], (x_search_l[i], x_search_r[i]), FalsePosition(5), verbose=true)
s = sign(f[i](prevfloat((x)))*sign(f[i](nextfloat(x))))
if s <= 0
    changes_sign = true
else
    changes_sign = false
end
println(f[i](x),"\t" ,iszero(f[i](x)), "\t", changes_sign)
end
```

Rezultat był następujący:

```
e^x-1/(10*x)^2
Results of univariate zero finding:

* Converged to: 0.09534461720025875
* Algorithm: FalsePosition{5}()
* iterations: 16
* function evaluations: 18
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
(a_0, b_0) = ( 0.0000000000000000,  4.0000000000000000)
(a_1, b_1) = ( 0.0000000000000000,  2.0000000000000000)
(...)
(a_15, b_15) = ( 0.0953446172045514,  0.0953446171948915)
(a_16, b_16) = ( 0.0953446171948915,  0.0953446172002587)

2.220446049250313e-16        false        true
---------------------------------
cos(x)-x
Results of univariate zero finding:

* Converged to: 0.7390851332151607
* Algorithm: FalsePosition{5}()
* iterations: 7
* function evaluations: 9
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
(a_0, b_0) = ( 0.0000000000000000,  4.0000000000000000)
(a_1, b_1) = ( 4.0000000000000000,  0.7075083376742781)
(..)
(a_6, b_6) = ( 0.7390851845804344,  0.7390851332151601)
(a_7, b_7) = ( 0.7390851845804344,  0.7390851332151607)

0.0        true        true
---------------------------------
20*x/(100*x^2+1)
Results of univariate zero finding:

* Converged to: 3.0
* Algorithm: FalsePosition{5}()
* iterations: 0
* function evaluations: 3
```

```
* stopped as x_n = x_{n-1} using atol=xatol, rtol=xrtol
* Note: Exact zero found

Trace:
(a_0, b_0) = (-2.0000000000000000,  3.0000000000000000)

0.0        true        true
---------------------------------
+/-(|x|^(1/3))*e^(-x^2)
Results of univariate zero finding:

* Converged to: 3.0
* Algorithm: FalsePosition{5}()
* iterations: 0
* function evaluations: 3
* stopped as x_n = x_{n-1} using atol=xatol, rtol=xrtol
* Note: Exact zero found

Trace:
(a_0, b_0) = (-2.0000000000000000,  3.0000000000000000)

0        true        true
---------------------------------
pi*(x-5)/180 - 0.8*sin(pi*x/180)
Results of univariate zero finding:

* Converged to: 22.656578669567754
* Algorithm: FalsePosition{5}()
* iterations: 8
* function evaluations: 10
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
(a_0, b_0) = ( 0.0000000000000000,  45.0000000000000000)
(a_1, b_1) = ( 45.0000000000000000,  17.8732960351904886)
(...)
(a_7, b_7) = ( 22.6567890424722087,  22.6565786692063611)
(a_8, b_8) = ( 22.6567890424722087,  22.6565786695677538)

0.0        true        true
---------------------------------
cos(100x)-4*erf(30x-10)
Results of univariate zero finding:

* Converged to: 0.33186603357456257
* Algorithm: FalsePosition{5}()
* iterations: 11
* function evaluations: 13
* stopped as x_n = x_{n-1} using atol=xatol, rtol=xrtol
* Note: x_n = x_{n-1}. Change of sign at xn identified.
        Algorithm stopped early, but |f(xn)| < e^(1/3), where e depends on xn, rtol, and atol.

Trace:
(a_0, b_0) = ( 0.0000000000000000,  0.6000000000000000)
(a_1, b_1) = ( 0.6000000000000000,  0.3014344366440125)
(...)
(a_10, b_10) = ( 0.3318660340164839,  0.3318660335745625)
(a_11, b_11) = ( 0.3318660335745625,  0.3318660335745626)

-7.91033905045424e-15        false        true
```

1.3 Metoda Newtona

Ponowiłem obliczenia przy użyciu metody Newtona z wykorzystaniem pochodnych funkcji:

```
for i in 1:length(f)
    println(labels[i])
    x = find_zero((f[i], D(f[i])), start[i], Roots.Newton(), verbose=true)
    s = sign(f[i](prevfloat((x)))*sign(f[i](nextfloat(x))))
    if s <= 0
        changes_sign = true
    else
        changes_sign = false
    end
    println(f[i](x),"\t" ,iszero(f[i](x)), "\t", changes_sign)
end
```

Otrzymany wynik:

```
e^x-1/(10*x)^2
Results of univariate zero finding:

* Converged to: 0.09534461720025875
* Algorithm: Roots.Newton()
* iterations: 12
* function evaluations: 25
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  1.0000000000000000,         fx_0 =  2.7082818284590453
x_1 =  0.0109557751463742,         fx_1 = -82.3021770135689508
(...)
x_11 =  0.0953446172002576,         fx_11 = -0.0000000000000275
x_12 =  0.0953446172002587,         fx_12 =  0.0000000000000002

2.220446049250313e-16        false        true
----------------------------------
cos(x)-x
Results of univariate zero finding:

* Converged to: 0.7390851332151607
* Algorithm: Roots.Newton()
* iterations: 4
* function evaluations: 9
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  1.0000000000000000,         fx_0 = -0.4596976941318602
x_1 =  0.7503638678402439,         fx_1 = -0.0189230738221174
x_2 =  0.7391128909113617,         fx_2 = -0.0000464558989908
x_3 =  0.7390851333852840,         fx_3 = -0.0000000002847206
x_4 =  0.7390851332151607,         fx_4 =  0.0000000000000000

0.0        true        true
----------------------------------
20*x/(100*x^2+1)
Results of univariate zero finding:

* Converged to: 1.7002120211960927e7
* Algorithm: Roots.Newton()
* iterations: 24
* function evaluations: 49
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol
```

```
Trace:
x_0 =  1.0000000000000000,         fx_0 =  0.1980198019801980
x_1 =  2.0202020202020203,         fx_1 =  0.0987580181659888
(...)
x_23 =  8501060.1059804614633322,         fx_23 =  0.0000000235264776
x_24 =  17002120.2119609266519547,         fx_24 =  0.0000000117632388

1.176323879061276e-8        false        false
---------------------------------
+/-(|x|^(1/3))*e^(-x^2)
Results of univariate zero finding:

* Converged to: -5.817879689923602
* Algorithm: Roots.Newton()
* iterations: 31
* function evaluations: 63
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  0.3000000000000000,         fx_0 =  0.6118156495282121
x_1 = -1.6565217391304339,         fx_1 = -0.0760910788611134
(...)
x_30 = -5.7301771844857070,         fx_30 = -0.0000000000000098
x_31 = -5.8178796899236023,         fx_31 = -0.0000000000000036

-3.5895646777142075e-15        false        false
---------------------------------
pi*(x-5)/180 - 0.8*sin(pi*x/180)
Results of univariate zero finding:

* Converged to: 22.656578669567754
* Algorithm: Roots.Newton()
* iterations: 5
* function evaluations: 11
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  1.0000000000000000,         fx_0 = -0.0837750952296000
x_1 =  24.9851846856033255,         fx_1 =  0.0109001558564727
(...)
x_4 =  22.6565786695788134,         fx_4 =  0.0000000000000505
x_5 =  22.6565786695677538,         fx_5 =  0.0000000000000000

0.0        true        true
---------------------------------
cos(100x)-4*erf(30x-10)
Results of univariate zero finding:

* Converged to: 0.3318660335745625
* Algorithm: Roots.Newton()
* iterations: 6
* function evaluations: 13
* stopped as x_n = x_{n-1} using atol=xatol, rtol=xrtol
* Note: x_n = x_{n-1}. Change of sign at xn identified.
        Algorithm stopped early, but |f(xn)| < e^(1/3), where e depends on xn, rtol, and atol.

Trace:
x_0 =  0.3200000000000000,         fx_0 =  2.5477927806931855
x_1 =  0.3349406455162889,         fx_1 = -0.7033258478870961
(...)
x_5 =  0.3318660335745626,         fx_5 = -0.0000000000000079
x_6 =  0.3318660335745625,         fx_6 =  0.0000000000000071

7.077671781985373e-15        false        true
---------------------------------
```

1.4 Metoda Steffensena

Powtórzyłem obliczenia przy użyciu metody Steffensa:

```
for i in 1:length(f)
    println(labels[i])
    x = find_zero(f[i], start[i], Order2(), verbose=true)
    s = sign(f[i](prevfloat((x)))*sign(f[i](nextfloat(x))))
    if s <= 0
        changes_sign = true
    else
        changes_sign = false
    end
    println(f[i](x),"\t" ,iszero(f[i](x)), "\t", changes_sign)
end
```

Co dało rezultat:

```
e^x-1/(10*x)^2
Results of univariate zero finding:

* Converged to: -8.999510577041852
* Algorithm: Order2()
* iterations: 23
* function evaluations: 34
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  1.0000000000000000,         fx_0 =  2.7082818284590453
x_1 =  0.0109586822408080,         fx_1 = -82.2579776666424891
(...)
x_22 = -8.9995105768317831,         fx_22 =  0.0000000000000207
x_23 = -8.9995105770418515,         fx_23 =  0.0000000000000005

4.9203805092823405e-16        false        false
----------------------------------
cos(x)-x
Results of univariate zero finding:

* Converged to: 0.7390851332151603
* Algorithm: Order2()
* iterations: 4
* function evaluations: 8
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  1.0000000000000000,         fx_0 = -0.4596976941318602
x_1 =  0.7503640896092518,         fx_1 = -0.0189234468165282
x_2 =  0.7396466488727576,         fx_2 = -0.0009398758559590
x_3 =  0.7390850863084190,         fx_3 =  0.0000000785036863
x_4 =  0.7390851332151603,         fx_4 =  0.0000000000000004

4.440892098500626e-16        false        false
----------------------------------
20*x/(100*x^2+1)
Results of univariate zero finding:

* Converged to: 2.3314039945266686e7
* Algorithm: Order2()
* iterations: 26
* function evaluations: 48
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol
```

```
Trace:
x_0 =  1.0000000000000000,         fx_0 =  0.1980198019801980
x_1 =  2.0202080132910138,         fx_1 =  0.0987577266255866
(...)
x_25 =  11142472.8398810550570488,         fx_25 =  0.0000000179493370
x_26 =  23314039.9452666863799095,         fx_26 =  0.0000000085785218

8.578521803579771e-9        false        false
---------------------------------
+/-(|x|^(1/3))*e^(-x^2)
Results of univariate zero finding:

* Converged to: -5.858316025427789
* Algorithm: Order2()
* iterations: 34
* function evaluations: 63
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  0.3000000000000000,         fx_0 =  0.6118156495282121
x_1 = -1.6565848195572388,         fx_1 = -0.0760761437191582
(...)
x_33 = -5.7647567514216780,         fx_33 = -0.0000000000000066
x_34 = -5.8583160254277891,         fx_34 = -0.0000000000000022

-2.2438589727829635e-15        false        false
---------------------------------
pi*(x-5)/180 - 0.8*sin(pi*x/180)
Results of univariate zero finding:

* Converged to: 22.65657866956753
* Algorithm: Order2()
* iterations: 5
* function evaluations: 11
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  1.0000000000000000,         fx_0 = -0.0837750952296000
x_1 =  24.9851845841122788,         fx_1 =  0.0109001553695874
(...)
x_4 =  22.6565786695786215,         fx_4 =  0.0000000000000496
x_5 =  22.6565786695675300,         fx_5 = -0.0000000000000010

-9.992007221626409e-16        false        false
---------------------------------
cos(100x)-4*erf(30x-10)
Results of univariate zero finding:

* Converged to: 0.33186603357456257
* Algorithm: Order2()
* iterations: 7
* function evaluations: 13
* stopped as x_n = x_{n-1} using atol=xatol, rtol=xrtol
* Note: x_n = x_{n-1}. Change of sign at xn identified.
        Algorithm stopped early, but |f(xn)| < e^(1/3), where e depends on xn, rtol, and atol.

Trace:
x_0 =  0.3200000000000000,         fx_0 =  2.5477927806931855
x_1 =  0.3349376989005570,         fx_1 = -0.7026702179875287
(...)
x_6 =  0.3318660335745625,         fx_6 =  0.0000000000000071
x_7 =  0.3318660335745626,         fx_7 = -0.0000000000000079

-7.91033905045424e-15        false        true
```

1.5 Podsumowanie wyników uzyskanych za pomocą trzech metod:

Na postawie otrzymanych wyników zebrałem dane statystyczne wykonanych obliczeń i umieściłem w tabeli 1:

Funkcja	**False Position**				**Newton**				**Steffensen**			
	Iteracje	**Wywołania**	**Dokładne (zero)**	**Zmiana znaku**	**Iteracje**	**Wywołania**	**Dokładne (zero)**	**Zmiana znaku**	**Iteracje**	**Wywołania**	**Dokładne (zero)**	**Zmiana znaku**
$\mathbf{e^x - \frac{1}{(10x)^2}}$	16	18		+	12	25		+	23	34		
$\mathbf{\cos(x) - x}$	7	9	+	+	4	9	+	+	4	8		
$\mathbf{\frac{20x}{100x^2+1}}$	0	3	+	+	24	49			26	48		
$\mathbf{\pm \lvert x \rvert^{\frac{1}{3}} \cdot e^{-x^2}}$	0	3	+	+	31	63			34	63		
$\mathbf{\pi \cdot \frac{x-5}{180} - 0.8 \cdot \sin\left(\frac{x\pi}{180}\right)}$	8	10	+	+	5	11	+	+	5	11		
$\mathbf{\cos(100x) - 4 \cdot erf(30x - 10)}$	11	13		+	6	13		+	7	13		+
Średnia	7.0	9.3			13.7	28.3			16.5	29.5		

Tabela 1: Podsumowanie liczby wywołań funkcji i dokładności obliczonych danych

1.6 Wnioski

Otrzymane wyniki wskazują, że różne metody obliczania pierwiastków mogą dawać różne wyniki. Na tabeli 1 wyraźnie widać, że metoda korzystająca ze zmiany znaku potrzebowała najmniej iteracji do otrzymania prawidłowego wyniku. 0 iteracji w 3. i 4. przypadku jest najprawdopodobniej wynikiem faktu, że obie te badane funkcje są nieparzyste i zerują się w 0, co pozwoliło algorytmowi na błyskawiczne odnalezienie miejsca zerowego.

W kolumnie dokładne oznaczyłem znakiem "+" te wyniki, dla których wartość funkcji faktycznie była równa 0. Zmiana znaku zaś oznacza, że najbliższa liczba na lewo do wyniku jest przeciwnego znaku niż najbliższa liczba z prawej.

Dokładne wyniki otrzymane za pomocą pierwszej z badanych metod są również efektem nieprzypadkowego wyboru użytej metody. Spośród 12 implementacji wybrałem tą, która dla badanych funkcji dawała najlepsze wyniki - wersję 5. W sekcji nr 5 opisuję porównanie wyników różnych implementacji metody False Position dla badanych funkcji.

2 Demonstracja działania metod poszukiwania pierwiastków na wybranej funkcji

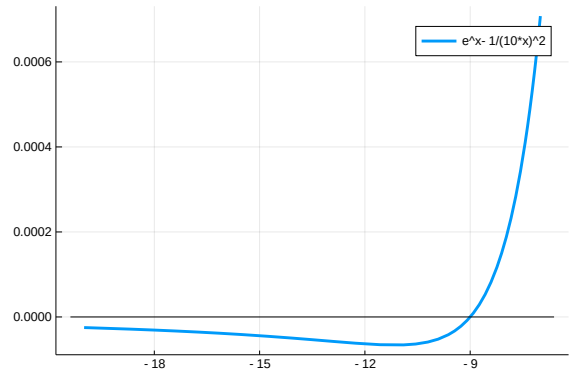
Zadanie: Zademonstrować wybrany, ciekawy przykład trudnej funkcji z p.1 i działania metod na niej.

2.1 Badana funkcja

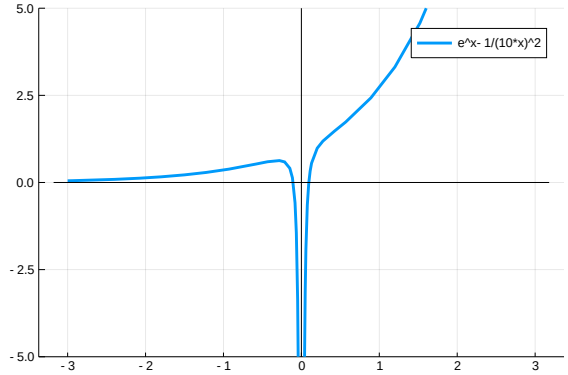
Do dokładniejszej analizy wybrałem funkcję:

$$f(x) = e^x - \frac{1}{(10x)^2} \tag{2}$$

Przyczyną wybory właśnie tej była obecność trzech miejsc zerowych oraz jej nieciągłość w 0. Wykres przedstawiłem na rysunku 2.



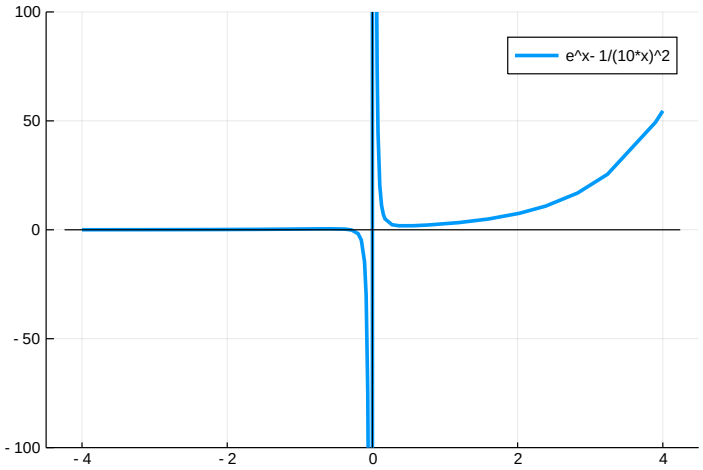
(a) Wykres z widocznym pierwiastkiem funkcji ≈ -9



(b) Wykres funkcji w okolicach 0

Rysunek 2: Wykres funkcji na przedziałach w których występują jej pierwiastki

Dodatkowo sporządziłem wykres funkcji pochodnej - widoczny na rys. 3.



Rysunek 3: Wykres pochodnej badanej funkcji

2.2 Wyniki działania różnych metod obliczania pierwiastków

Przy użyciu poniższego kodu przeprowadziłem obliczenia przy użyciu 6 różnych metod obliczania pierwiastków. Aby otrzymane wyniki dotyczyły tego samego pierwiastka (funkcja ma ich 3) zmodyfikowałem odpowiednio punkt startowy poszukiwań w wymagających tego metodach.

```
x = find_zero(f[1], (x_search_l[1], x_search_r[1]), Bisection(), verbose=true)
x = find_zero(f[1], (x_search_l[1], x_search_r[1]), FalsePosition(5), verbose=true)
# Newton
x = find_zero((f[1], D(f[1])), start[1]-0.9, Roots.Newton(), verbose=true)
# Halley
x = find_zero((f[1], D(f[1]), D(D(f[1]))), start[1]-0.9, Roots.Halley(), verbose=true)
# Metoda siecznych
x = find_zero(f[1], start[1]-0.9, Order1(), verbose=true)
# Steffensen
x = find_zero(f[1], start[1]-0.9, Order2(), verbose=true)
```

Szczegółowy rezultat wykonania:

```
Results of univariate zero finding:

* Converged to: 0.09534461720025875
* Algorithm: Roots.BisectionExact()
* iterations: 62
* function evaluations: 64
* stopped as x_n = x_{n-1} using atol=xatol, rtol=xrtol
* Note: Change of sign at xn identified.

Trace:
(a_0, b_0) = ( 0.0000000000000000,  4.0000000000000000)
(a_1, b_1) = ( 0.0000000000000000,  4.0000000000000000)
(...)
(a_13, b_13) = ( 0.0934143066406250,  0.1012344360351563)
(a_14, b_14) = ( 0.0934143066406250,  0.0973243713378906)
(...)
(a_61, b_61) = ( 0.0953446172002587,  0.0953446172002588)
(a_62, b_62) = ( 0.0953446172002587,  0.0953446172002587)

Results of univariate zero finding:

* Converged to: 0.09534461720025875
* Algorithm: FalsePosition{5}()
* iterations: 16
* function evaluations: 18
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
(a_0, b_0) = ( 0.0000000000000000,  4.0000000000000000)
(a_1, b_1) = ( 0.0000000000000000,  2.0000000000000000)
(...)
(a_15, b_15) = ( 0.0953446172045514,  0.0953446171948915)
(a_16, b_16) = ( 0.0953446171948915,  0.0953446172002587)

Results of univariate zero finding:

* Converged to: 0.09534461720025875
* Algorithm: Roots.Newton()
* iterations: 4
* function evaluations: 9
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol
```

```
Trace:
x_0 =  0.1000000000000000,         fx_0 =  0.1051709180756473
x_1 =  0.0950168175143479,         fx_1 = -0.0079636939147820
x_2 =  0.0953430071258671,         fx_2 = -0.0000389245275041
x_3 =  0.0953446171613899,         fx_3 = -0.0000000009396552
x_4 =  0.0953446172002587,         fx_4 =  0.0000000000000002

Results of univariate zero finding:

* Converged to: 0.09534461720025877
* Algorithm: Roots.Halley()
* iterations: 2
* function evaluations: 9
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  0.1000000000000000,         fx_0 =  0.1051709180756473
x_1 =  0.0953458783641774,         fx_1 =  0.0000304880920805
x_2 =  0.0953446172002588,         fx_2 =  0.0000000000000007

Results of univariate zero finding:

* Converged to: 0.09534461720025875
* Algorithm: Roots.Secant()
* iterations: 6
* function evaluations: 8
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  0.1000000000000000,         fx_0 =  0.1051709180756473
x_1 =  0.0950163893725045,         fx_1 = -0.0079741467693415
x_2 =  0.0953676202692240,         fx_2 =  0.0005559080165574
x_3 =  0.0953447303727287,         fx_3 =  0.0000027359425878
x_4 =  0.0953446171612243,         fx_4 = -0.0000000009436580
x_5 =  0.0953446172002588,         fx_5 =  0.0000000000000018
x_6 =  0.0953446172002587,         fx_6 =  0.0000000000000002

Results of univariate zero finding:

* Converged to: 0.09534461720025875
* Algorithm: Order2()
* iterations: 5
* function evaluations: 10
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  0.1000000000000000,         fx_0 =  0.1051709180756473
x_1 =  0.0950163893725045,         fx_1 = -0.0079741467693415
x_2 =  0.0953676202692240,         fx_2 =  0.0005559080165574
x_3 =  0.0953448009075610,         fx_3 =  0.0000044411162956
x_4 =  0.0953446172119856,         fx_4 =  0.0000000002834968
x_5 =  0.0953446172002587,         fx_5 =  0.0000000000000002
```

Wyniki zebrałem i porównałem w tabeli 2.

Metoda:	x	f(x)	Zero	Zmiana znaku	Iteracje	Wywołania funkcji
Bisekcja:	0.095(...)875	2.22E-16	-	+	62	64
False position:	0.095(...)875	2.22E-16	-	+	16	18
Newton:	0.095(...)875	2.22E-16	-	+	4	9
Halley:	0.095(...)877	6.66E-16	-	-	2	9
Siecznych:	0.095(...)875	2.22E-16	-	+	6	8
Steffensen:	0.095(...)875	2.22E-16	-	+	5	10

Tabela 2: Wyniki działania różnych metod wyznaczania pierwiastków funkcji

Co warte zauważenia - wszystkie metody poza metodą Halleya odnalazły pierwiastek z tą samą dokładnością, choć metody nie korzystające z pochodnych potrzebowały na to znacznie więcej iteracji, co było wynikiem m.in. szerszego przedziału początkowego.

2.3 Wyniki działania 12 metod obliczania pierwiastków metodą *False Position*

Następnie przetestowałem wszystkie 12 metod wyznaczania pierwiastków metodą False Position. Wyniki zgromadziłem w tabeli 3. Do obliczeń użyłem poniższego kodu:

```
for j in 1:12
    x = find_zero(f[1], (x_search_l[1], x_search_r[1]), FalsePosition(j), verbose=false)
    s = sign(f[1](prevfloat((x)))*sign(f[1](nextfloat(x))))
    if s <= 0
        changes_sign = true
    else
        changes_sign = false
    end
    println(j,"\t", x,"\t" ,f[1](x),"\t",iszero(f[1](x)), "\t",
    sign(f[1](prevfloat((x)))*sign(f[1](nextfloat(x))))<=0)
end
```

Numer metody	x	f(x)	Zero	Zmiana znaku	Iteracje	Wywołania funkcji
1	0.095(...)5875	2.22E-16	-	+	14	16
2	0.095(...)5875	2.22E-16	-	+	15	17
3	0.095(...)5877	6.66E-16	-	-	14	16
4	0.095(...)5876	4.44E-16	-	-	13	15
5	0.095(...)5875	2.22E-16	-	+	16	18
6	0.095(...)5875	2.22E-16	-	+	17	19
7	0.095(...)5875	2.22E-16	-	+	17	19
8	0.095(...)5875	2.22E-16	-	+	15	17
9	0.095(...)5875	2.22E-16	-	+	14	16
10	0.095(...)5875	2.22E-16	-	+	16	18
11	0.095(...)5877	6.66E-16	-	-	16	18
12	0.095(...)5876	4.44E-16	-	-	13	15

Tabela 3: Wyniki dwunastu metod False Position

Tylko 8 z 12 metod uzyskało wynik najbliższy dokładnemu. Wszystkie wykonały natomiast bardzo podobną liczbę iteracji.

2.4 Wszystkie pierwiastki funkcji

Na zakończenie wygenerowałem wszystkie pierwiastki funkcji przy użyciu metody `find_zeros(f[1], -10, 10)`

```
        x                    f(x)        is_zero?     change_sign?
-8.999510577046975    5.421010862427522e-20     false        true
-0.1054119671030927 -1.1102230246251565e-16     false        true
0.09534461720025873  -4.440892098500626e-16     false        true
```

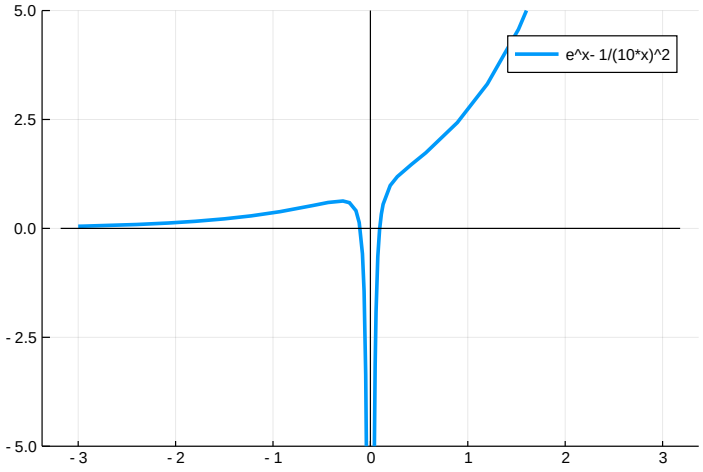
3 Demonstracja przykładów nieprawidłowego działania

Zadanie: Zademonstrować wybrany, ciekawy przykład trudnej funkcji z p.1 i działania metod na niej.

3.1 Metoda *False Position*

Metoda *False Position* nie zadziała, jeżeli krańce przedziału, który chcemy przeszukiwać są tego samego znaku. Przykładowo dla wywołania metody na pierwszej z badanych funkcji: $f(x) = e^x - \frac{1}{(10x)^2}$ przestawionej na rys. 4 na przedziale $[-2, 2]$ otrzymamy komunikat o błędzie.

```
ArgumentError: The interval [a,b] is not a bracketing interval.
You need f(a) and f(b) to have different signs (f(a) * f(b) < 0).
Consider a different bracket or try fzero(f, c) with an initial guess c.
```



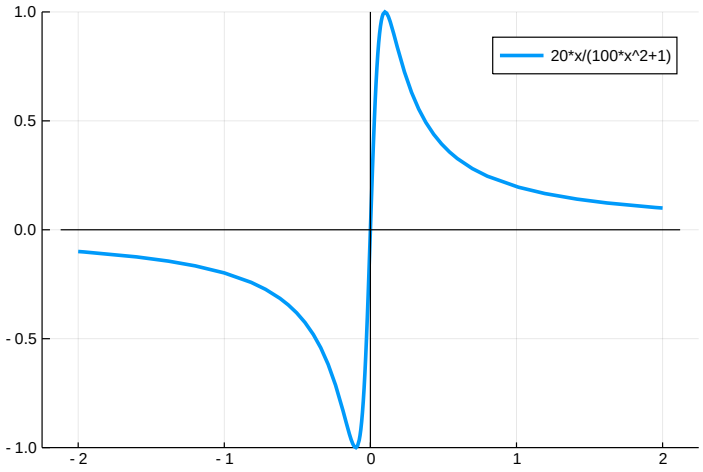
Rysunek 4: Wykres funkcji, która generuje problemy

Przyczyną jest fakt, że zarówno $f(2)$ jak i $f(-2)$ są dodatnie. Rozwiązaniem jest wybranie punktów o przeciwnych co do znaku wartościach funkcji. Na przykład w celu odnalezienia dodatniego minimum - przedział od 0.01 do 1.

3.2 Metoda Newtona

Metoda Newtona często nie zadziała, jeżeli funkcja nie jest monotoniczna, a wybierzemy punkt startowy daleko od miejsca zerowego. Jeżeli funkcja zmieniła monotoniczność pomiędzy punktem startowym, a pierwiastkiem może się okazać, że oddalamy się dod miejsca serowego. Przykładowo dla wywołania metody na trzeciej z badanych funkcji: $f(x) = \pi \cdot \frac{x-5}{180} - 0.8 \cdot \sin\left(\frac{x\pi}{180}\right)$ przestawionej na rys. 5 z punktem startowym 1.0 otrzymamy komunikat o znalezieniu miejsca zerowego dla bardzo dużej wartości.

```
Results of univariate zero finding:

* Converged to: 1.7002120211960927e7
* Algorithm: Roots.Newton()
* iterations: 24
* function evaluations: 49
* stopped as |f(x_n)| <= max(d, max(1,|x|)*e) using d = atol, e = rtol

Trace:
x_0 =  1.0000000000000000,          fx_0 =  0.1980198019801980
x_1 =  2.0202020202020203,          fx_1 =  0.0987580181659888
x_2 =  4.0503283574619111,          fx_2 =  0.0493486314602615
(...)
x_23 =  8501060.1059804614633322,          fx_23 =  0.0000000235264776
x_24 =  17002120.2119609266519547,          fx_24 =  0.0000000117632388
```



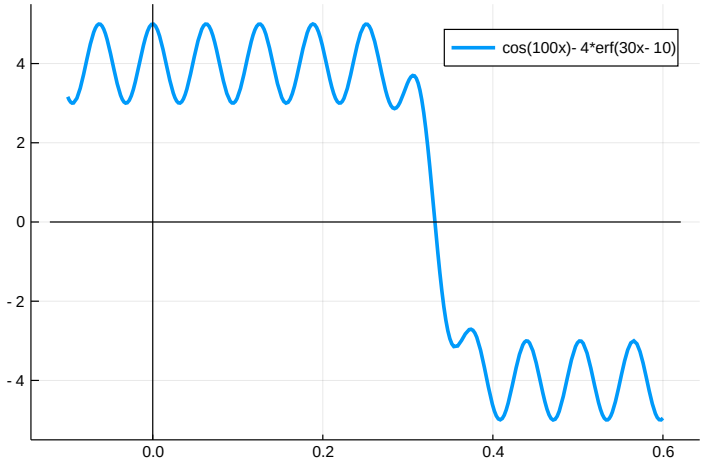
Rysunek 5: Wykres funkcji, która generuje problemy dla metody Newtona

Jest to spowodowane faktem, że funkcja asymptotycznie dąży do 0 w nieskończoności. Nie udało się natomiast odnaleźć pierwiastka w 0. Każda kolejna iteracja oddalała się od niego. Rozwiązaniem jest wybieranie punktu startowego bliżej szukanego pierwiastka.

3.3 Metoda Steffensena

Metoda Steffensena również często nie działa dla funkcji niemonotonicznych.Zachodzi tu podobny mechanizm, co dla metody Newtona. Przykładowo dla wywołania metody na szóstej z badanych funkcji: $f(x) = \cos(100x) - 4 \cdot \mathrm{erf}(30x - 10)$ przestawionej na rys. 6 z punktem startowym 0.3 otrzymamy komunikat o braku zbieżności:

```
Roots.ConvergenceFailed("Stopped at: xn = -143314.4170861863")
```



Rysunek 6: Wykres funkcji, która generuje problemy dla metody Steffensena

Śledząc kolejne wywołania obserwujemy oddalanie się badanego punku w niekontrolowany sposób od pierwiastka:

```
Trace:
x_0 =  0.3000000000000000,        fx_0 =  3.5250546216864436
x_1 =  0.2280256452124042,        fx_1 =  3.3114962022096508
x_2 = -0.8880287024435723,        fx_2 =  4.6687456397881792
x_3 =  2.9510400237013261,        fx_3 = -3.0210831958901538
x_4 =  1.4427950075558296,        fx_4 = -3.0271979844485957
x_5 =  748.1172268858236976,        fx_5 = -4.5596535402085685
x_6 = -1473.5307053496881053,        fx_6 =  4.9817599789555080
x_7 = -313.5645500299915511,        fx_7 =  3.0238256478786925
x_8 =  1477.8823618783765141,        fx_8 = -3.8642022172105612
x_9 =  472.8758107148921681,        fx_9 = -3.0534564153315635
x_10 = -3312.2115668431779341,        fx_10 =  3.0048350476449528
x_11 = -3331.6638612591777928,        fx_11 =  4.8845757571551349
x_12 = -3281.1163547628184460,        fx_12 =  3.151230596721480
```

Funkcja ma bardzo wiele przedziałów monotoniczności, które powodują tak duże "skoki" w kolejnych iteracjach. Podobnie jak w poprzednim przykładzie rozwiązaniem jest lepsze dobieranie punktu startowego.

4 Wstęga Newtona

Zadanie: Narysować wstegę Newtona i objaśnić, w jaki sposób powstała i jaki jest jej związek z metodą Newtona do znajdowania pierwiastków. Sposób i język - dowolny.

4.1 Program

Do realizacji zadania wybrałem język Python z uwagi na bogate i proste w użyciu biblioteki numeryczne, oraz możliwość łatwego generowania wykresów.

Użyty kod:

```
import numpy as np
import math
import matplotlib.pyplot as plt


def f(z):
    return z**3 - 2*z + 2


def calculate_newton_fractal(width_px, height_px, niter):

    output_pix = np.arange(width_px*height_px*3, dtype=np.uint32).reshape(height_px, width_px, 3)
    x_min, x_max, y_min, y_max = -3, 3, -3, 3
    h = 1e-6
    max_err = 1e-8
    root1 = -1.76929235423863
    root2 = complex(0.884646177119316, -0.589742805022206)
    root3 = complex(0.884646177119316, 0.589742805022206)

    color_mult = 8
    for y in range(height_px):
        zy = y * (y_max - y_min) / (height_px - 1) + y_min

        for x in range(width_px):
            zx = x * (x_max - x_min) / (width_px - 1) + x_min
            z = complex(zx, zy)
            count = 0
            for i in range(niter):
                der_z = (f(z + complex(h, h)) - f(z)) / complex(h, h)
                if der_z == 0:
                    break

                count += 1
                if count > 255:
                    break

                next_z = z - f(z) / der_z
                if abs(next_z - z) < max_err:
                    break

                z = next_z

            if abs(z-root1)<max_err:
                output_pix[y,x] = (255 - count*color_mult,
                50 - count*color_mult % 50, 120 - count*color_mult % 120)
            elif abs(z-root2)<=max_err:
                output_pix[y,x] = (120 - count*color_mult % 120,
                255 - count*color_mult, 120 - count*color_mult % 120)
            elif abs(z-root3)<=max_err:
                output_pix[y,x] = (120 - count*color_mult % 120,
                120 - count*color_mult % 120, 255 - count*color_mult)
```

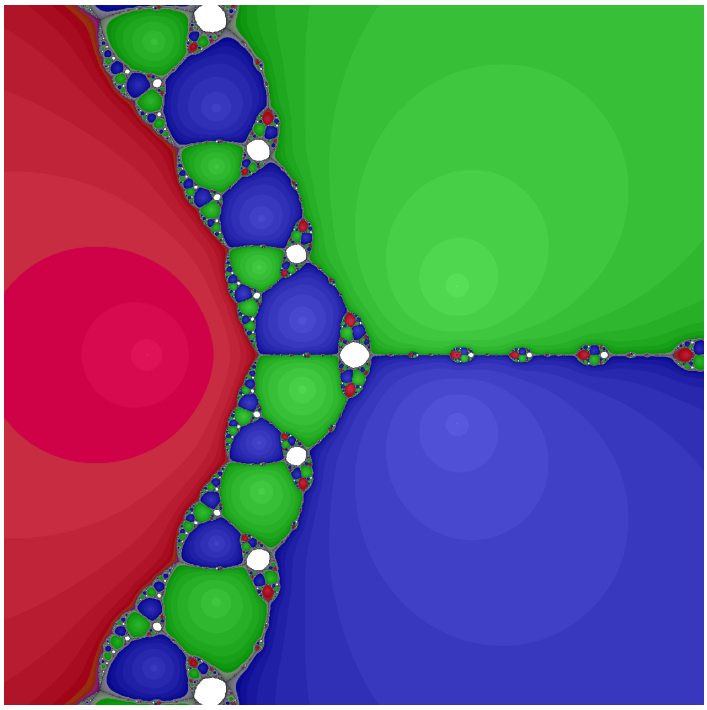
```
    return output_pix


def create_fractal(width=2048, height=2048, niter=4096):
    output_img = calculate_newton_fractal(width, height, niter=niter)
    plt.axis('off')
    plt.imshow(output_img)
    plt.savefig('newton_plot.pdf', dpi=400)


if __name__ == "__main__":
    create_fractal()
```

4.2 Przykładowy fraktal - wstęga Newtona

Do wygenerowania wstęgi Newtona użyłem funkcji:

$$f(z) = z^3 - 2z + 2 \tag{3}$$

Efekt widoczny jest na rysunku 7:



Rysunek 7: Fraktal Newtona

Powstawanie fraktala Dla każdego punktu na płaszczyźnie zespolonej stosując metodę Newtona dążymy do miejsca zerowego wybranej funkcji ($f(z) = z^3 - 2z + 2$). Kolor na wykresie oznacza, do którego pierwiastka zbiegliśmy, a jasność jest zależna od liczby iteracji przed osiągnięciem wystarczającej dokładności, (im ciemniej tym dłużej). Białe koła na wykresie to obszary, dla których nie udało się uzyskać zbieżności do któregoś z miejsc zerowych. W efekcie w zależności od użytej funkcji i sposobu kolorowania możemy uzyskać bardzo efektowne obrazy.

5 Dodatek - porównanie 12 implementacji metody False Position

Dodatkowo wykonałem testy każdej z 12 metod False Position na wszystkich 6 badanych fun kcjach, aby móc określić, która z metod jest w badanych przypadkach najlepsza. Użyłem do tego kodu:

```
using DataFrames
falsi = DataFrame(
    fun = String[],
    version = Int64[],
    is_zero = Bool[],
    sign_change = Bool[]
)
for i in 1:length(f)
    for j in 1:12
        x = find_zero(f[i], (x_search_l[i], x_search_r[i]), FalsePosition(j), verbose=false)
        s = sign(f[i](prevfloat((x)))*sign(f[i](nextfloat(x))))
        if s <= 0
            changes_sign = true
        else
            changes_sign = false
        end
        println(j, ".\t", f[i](x),"\t" ,iszero(f[i](x)), "\t", changes_sign)
        push!(falsi, [labels[i], j, iszero(f[i](x)), changes_sign ])
    end
    println("---------------------------------" )
end
```

W efekcie otrzymałem następujące dokładności:

```
1.      2.220446049250313e-16       false       true
2.      2.220446049250313e-16       false       true
3.      6.661338147750939e-16       false       false
4.      4.440892098500626e-16       false       false
5.      2.220446049250313e-16       false       true
6.      2.220446049250313e-16       false       true
7.      2.220446049250313e-16       false       true
8.      2.220446049250313e-16       false       true
9.      2.220446049250313e-16       false       true
10.      2.220446049250313e-16       false       true
11.      6.661338147750939e-16       false       false
12.      4.440892098500626e-16       false       false
---------------------------------
1.      0.0     true    true
2.      4.440892098500626e-16       false       false
3.      6.661338147750939e-16       false       false
4.      0.0     true    true
5.      0.0     true    true
6.      0.0     true    true
7.      0.0     true    true
8.      6.661338147750939e-16       false       false
9.      0.0     true    true
10.      0.0     true    true
11.      0.0     true    true
12.      0.0     true    true
---------------------------------
1.      0.0     true    true
2.      0.0     true    true
3.      0.0     true    true
```

```
4.        0.0        true        true
5.        0.0        true        true
6.        0.0        true        true
7.        0.0        true        true
8.        0.0        true        true
9.        0.0        true        true
10.        0.0        true        true
11.        0.0        true        true
12.        0.0        true        true
---------------------------------
1.        0        true        true
2.        0        true        true
3.        0        true        true
4.        0        true        true
5.        0        true        true
6.        0        true        true
7.        0        true        true
8.        0        true        true
9.        0        true        true
10.        0        true        true
11.        0        true        true
12.        0        true        true
---------------------------------
1.        -2.9976021664879227e-15        false        false
2.        -6.827871601444713e-15        false        false
3.        -5.551115123125783e-17        false        false
4.        0.0        true        true
5.        0.0        true        true
6.        -1.609823385706477e-15        false        false
7.        -2.831068712794149e-15        false        false
8.        5.551115123125783e-17        false        true
9.        -5.551115123125783e-17        false        true
10.        -3.3306690738754696e-16        false        false
11.        0.0        true        true
12.        0.0        true        true
---------------------------------
1.        -7.91033905045424e-15        false        true
2.        7.077671781985373e-15        false        true
3.        -7.91033905045424e-15        false        true
4.        -7.91033905045424e-15        false        true
5.        -7.91033905045424e-15        false        true
6.        -7.91033905045424e-15        false        true
7.        1.50712775592865e-14        false        false
8.        -7.91033905045424e-15        false        true
9.        7.077671781985373e-15        false        true
10.        -2.2870594307278225e-14        false        false
11.        7.077671781985373e-15        false        true
12.        7.077671781985373e-15        false        true
---------------------------------
```

Kolejne kolumny to odpowiednio:

- numer metody
- f(x) w wyznaczonym punkcie
- czy jest 0?
- czy dla najbliższych sąsiadów zmienia znak?

Następnie dane przetworzyłem i zebrałem wyniki w tabeli 4. Na podstawie tych danych wybrałem metodę 5 do użycia we wcześniej opisanych zadaniach.

```
falsi_methods=by(falsi, :version,
    solved=:is_zero => sum,
    changed_sign=:sign_change => sum,
)
sort!(falsi_methods, (:solved, :changed_sign), rev=(true,true))
```

	Numer metody	**Dokładne**	**Zmiana znaku**
1	**5**	4	6
2	**4**	4	5
3	**11**	4	5
4	**12**	4	5
5	**9**	3	6
6	**1**	3	5
7	**6**	3	5
8	**7**	3	4
9	**10**	3	4
10	**8**	2	5
11	**2**	2	4
12	**3**	2	3

Tabela 4: Metody uszeregowane od najbardziej skutecznych w przypadku badanych funkcji