

VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS

Laboratorinis darbas 1

Vienmatis optimizavimas

Nikita Gainulin

VILNIUS 2024

Turinys

1	Tikslo funkcija	2
2	Vienmačiai optimizavimo metodai ir jų algoritmai	2
2.1	Intervalo dalijimo pusiau metodas	2
2.2	Auksinio pjūvio metodas	3
2.3	Niutono metodas	5
3	Rezultatai ir jų palyginimai	7
3.1	Minimumo taškas ir funkcijos reikšmė tame taške	7
3.2	Greitis	7
3.3	Efektyvumas	8
4	Išvada	9
5	Priedas	9

1 Tikslo funkcija

Visi šioje ataskaitoje minėti metodai bus pritaikyti šiai tikslo funkcijai:

$$f(x) = \frac{(x^2 - 5)^2}{7} - 1 \quad (1)$$

Mano Python kode aprašyta taip:

```
1 def f(x):  
2     return ((x**2-5)**2)/7-1
```

2 Vienmačiai optimizavimo metodai ir jų algoritmai

2.1 Intervalo dalijimo pusiau metodas

Pirmiausia reikia nustatyti intervalą, kuriame ieškosime minimumo taško. Minėto intervalo ribas įsiminsime kintamaisiais l (kairioji riba) ir r (dešinioji riba). Pradiniame intervale pasirenkam tris bandymo taškus x_1 , x_2 ir x_m , taip, kad x_m būtų intervalo vidurio taškas, o x_1 ir x_2 tarp l ir vidurio taško bei r ir vidurio taško atitinkamai. Tada apskaičiuojame $f(x_m)$, $f(x_1)$ ir $f(x_2)$ reikšmes. Po to palyginame gautas reikšmes, t. y. kuriame taške funkcija įgyja didesnę reikšmę. Tas taškas ir nustato intervalą, kurį reikia atmesti. Kadangi ieškome minimumo taško, atmetamas atitinkamas intervalas, atitinkamai keičiami x_1 , x_2 ir x_m taškai bei perskaičiuojami $f(x_m)$, $f(x_1)$ ir $f(x_2)$ reikšmės. Algoritmas tęsiamas tol, kol gaunamas pakankamai tikslus minimumo taškas, t.y. kol intervalo ilgis $r - l$ nėra mažiau už tam tikrą ε reikšmę.

Štai kaip tai atrodo mano Python kode:

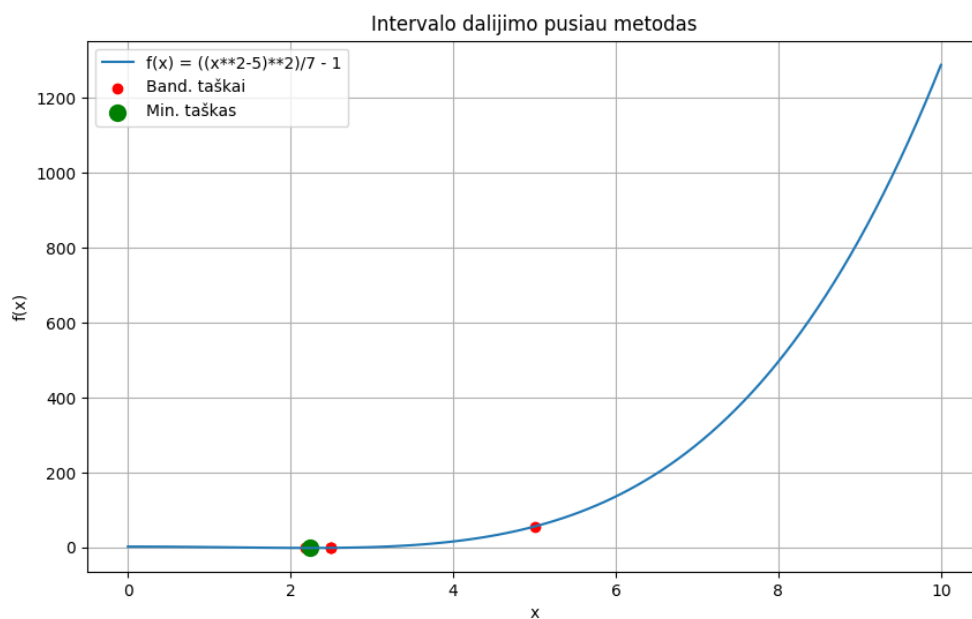
```
1 def halfCut(f, l, r, eps=1e-4):  
2     iterNum = 0  
3     funcNum = 0  
4  
5     xm = (l+r)/2  
6     fxm = f(xm)  
7     funcNum+=1  
8  
9     xVal = [xm]  
10    yVal = [fxm]  
11    while(r-l) > eps:  
12        iterNum += 1  
13        L = r-l  
14  
15        x1 = l+L/4  
16        x2 = r-L/4  
17  
18        fx1 = f(x1)  
19        fx2 = f(x2)  
20  
21        funcNum += 2  
22  
23        if fx1 < fxm:  
24            r = xm  
25            xm = x1  
26            fxm = fx1  
27            xVal.append(x1)  
28            yVal.append(fx1)
```

```

29
30     elif fx2 < fxm:
31         l = xm
32         xm = x2
33         fxm = fx2
34         xVal.append(x2)
35         yVal.append(fx2)
36
37     else:
38         l = x1
39         r = x2
40         xVal.append(xm)
41         yVal.append(fxm)
42
43     xm = (l+r)/2
44     xVal.append(xm)
45     yVal.append(f(xm))
46     return xm, f(xm), iterNum, funcNum, xVal, yVal

```

Vizualizuojant mūsų tikslo funkciją (1) šiuo metodu gauname štai tokį grafiką:



1 pav.: Intervalo dalijimo pusiau metodo vizualizacija (1) tikslo funkcijai.

2.2 Auksinio pjūvio metodas

Šis metodas yra labai panašus į intervalo dalijimo pusiau metodą, tačiau per kiekvieną iteraciją naudojami tik 2, o ne 3 bandomieji taškai. Mums vis dar reikia pasirinkti intervalą ir įsiminti jo kairiąją l ir dešiniąją r ribas, tačiau šį kartą x_1 ir x_2 apskaičiuojami kitaip, naudojant *Fibonačio*

reikšmę, kuri apytiksliai lygi:

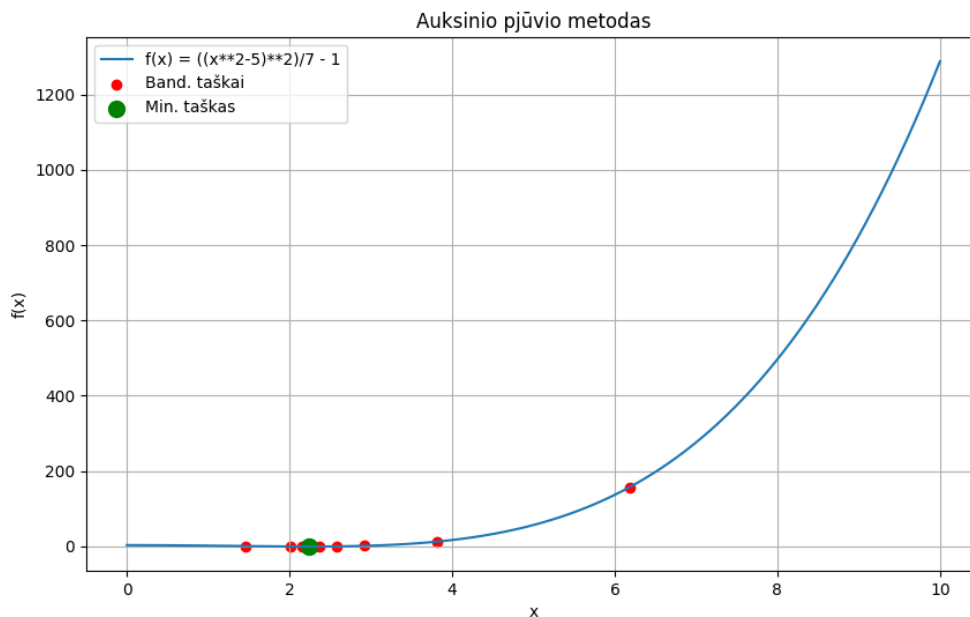
$$\tau = \frac{\sqrt{5}-1}{2} \approx 0.61803\dots$$

Apskaičiavę $x_1 = r - \tau(r - l)$ ir $x_2 = l + \tau(r - l)$ taškus, apskaičiuojame funkcijos reikšmes $f(x_1)$ ir $f(x_2)$ šiuose taškuose. Po to dar kartą palyginame funkcijos reikšmes ir pašaliname intervalus pagal didesnę vertę, perskaičiuojame taškus ir naujas funkcijos reikšmes ir kartojame tuos pačius veiksmus kiekvieną iteraciją, kol vėl gauname pakankamai tikslų minimumo tašką.

Algoritmo Python kodas:

```
1 def goldCut(f, l, r, eps=1e-4):
2     tau = (math.sqrt(5)-1)/2
3     print(tau)
4
5     L = r-l
6     x1 = l+(1-tau)*L
7     x2 = l+tau*L
8     fx1 = f(x1)
9     fx2 = f(x2)
10
11     iterNum = 0
12     funcNum = 2
13
14     xVal = [x1, x2]
15     yVal = [fx1, fx2]
16     while(r-l) > eps:
17         iterNum += 1
18         if fx1 < fx2:
19             r = x2
20             x2 = x1
21             fx2 = fx1
22             L = r-l
23             x1 = l+(1-tau)*L
24             fx1 = f(x1)
25             funcNum += 1
26             xVal.append(x1)
27             yVal.append(fx1)
28         else:
29             l = x1
30             x1 = x2
31             fx1 = fx2
32             L = r-l
33             x2 = l+tau*L
34             fx2 = f(x2)
35             funcNum += 1
36             xVal.append(x2)
37             yVal.append(fx2)
38
39     xm = (l+r)/2
40     xVal.append(xm)
41     yVal.append(f(xm))
42     return xm, f(xm), iterNum, funcNum, xVal, yVal
```

Vizualizuojant mūsų tikslo funkciją (1) šiuo metodu gauname štai tokį grafiką:



2 pav.: Auksinio pjūvio metodo vizualizacija (1) tikslo funkcijai.

2.3 Niutono metodas

Skirtingai nuo dviejų ankstesnių metodų, kurie remiasi konkrečiu intervalo pjūvimu, Niutono metodas remiasi vienu pradinio tašku x_i ir Teiloro eilutėmis iki antros eilės. Pasirinkus pradinį tašką x_i , kitas taškas apskaičiuojamas pagal šią formulę:

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)} \quad (2)$$

Tokiu būdu iteracija po iteracijos taškas artėja prie minimumo.

Vienas įdomus faktas, kurį pastebėjau, yra tai, kad daugelyje internetinių šaltinių kaip Niutono metodo pagrindas naudojama ši formulė:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Nors ši formulė paprastai naudojama dažniau ir bendriau, formulė (2) yra tikslesnė optimizavimo kontekste, todėl ją ir naudojame vietoj šios bendrosios formulės.

Algoritmo Python kodas:

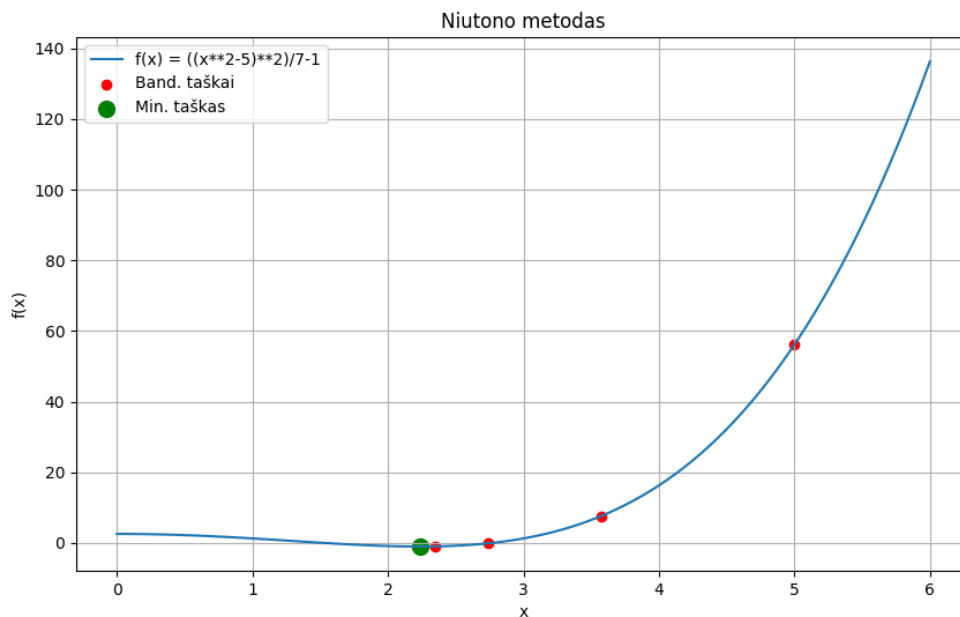
```
1 def newton(fSym, x0, eps=1e-4, maxIter=100):
2     x = sp.symbols('x')
3
4     fIsv = sp.diff(fSym, x)
5     fIsv2 = sp.diff(fIsv, x)
```

```

6
7     fSk = sp.lambdify(x, fSym)
8     fIsvSk = sp.lambdify(x, fIsv)
9     fIsv2Sk = sp.lambdify(x, fIsv2)
10
11     xi = x0
12     iterNum = 0
13     funcNum = 0
14
15     xVal = [xi]
16     yVal = [fSk(xi)]
17
18     while iterNum < maxIter:
19         fIsv_xi = fIsvSk(xi)
20         fIsv2_xi = fIsv2Sk(xi)
21         funcNum += 2
22
23         if abs(fIsv2_xi) < eps:
24             break
25
26         xiNext = xi - fIsv_xi / fIsv2_xi
27         iterNum += 1
28
29         xVal.append(xiNext)
30         yVal.append(fSk(xiNext))
31
32         if abs(xiNext - xi) < eps:
33             break
34
35         xi = xiNext
36
37     return xi, fSk(xi), iterNum, funcNum, xVal, yVal
38
39 x = sp.symbols('x')
40 fSym = ((x**2-5)**2)/7-1

```

Vizualizuojant mūsų tikslo funkciją (1) šiuo metodu gauname štai tokį grafiką:



3 pav.: Niutono metodo vizualizacija (1) tikslo funkcijai.

3 Rezultatai ir jų palyginimai

3.1 Minimumo taškas ir funkcijos reikšmė tame taške

Kaip ir galima tikėtis, taikant visus metodus gaunami daugiau ar mažiau panašūs rezultatai:

	Intervalo dalijimo pusiau metodas	Aukšinio metodas	pjūvio Niutono metodas
Minimumo taškas	2.236061	2.236057	2.236105
F-jos reikšmė	-0.999999	-0.999999	-0.999999

Matomas šiek tiek didesnis Niutono metodo minimumo taško nuokrypis nuo kitų metodų. Tačiau tai nebūtinai yra blogai, tiesiog rodo, kad Niutono metodo tikslumas priklauso nuo tokių parametrų kaip pradinis taškas ir funkcijos pobūdis, t.y. ar pati funkcija lygi, turi „staigių posūkių ar kampų“, kvadratinė, logaritminė ir t.t.

3.2 Greitis

Pirmiausia turėtume nustatyti tikslią greičio prasmę optimizavimo metodų algoritmų kontekste. Mūsų atveju **greitis** - tai kiekvieno algoritmo vidinių iteracijų kiekis, per kurį randamas tikslus minimumo taškas (arba intervalas, kuris yra mažesnis už iš anksto nustatytą ϵ). Paprastai kuo mažiau iteracijų algoritmui reikia minėtam mažiausiam taškui pasiekti, tuo jis yra greitesnis. Žinoma, tai

nebūtinai reiškia, kad minėtas algoritmas yra efektyvesnis, kaip pamatysime šiek tiek vėliau, tačiau griežtai kalbant tik apie greitį, atsižvelgiame tik į iteracijų skaičių.

Apskaičiavę kiekvieno algoritmo iteracijų skaičių mūsų tikslo funkcijai (1), gauname tokį rezultatą:

	Intervalo dalijimo pusiau metodas	Aukšinio pjūvio metodas	Niutono metodas
Iteracijų skaičius	17	24	6

Kaip matome, iš trijų metodų Niutono gali būti laikomas greičiausiu mūsų tikslo funkcijai (1). Tai nestebina, nes naudodamas pirmos ir antros eilės išvestines Niutono metodas minimumo taško link žengia daug didesniais žingsniais, priešingai nei intervalo dalijimo pusiau ir aukšinio pjūvio metodai, kurie, naudodami tam tikrus intervalus, prie jo artėja lėčiau. Skirtumas tarp kitų dviejų metodų taip pat gana tikėtinas, nes intervalo dalijimo pusiau metodas, kaip rodo pavadinimas, per kiekvieną iteraciją intervalą sumažina perpus, o aukšinio pjūvio metodas perkelia 61.8% ankstesnės iteracijos intervalo (nes $\tau \approx 0.61803\dots$), t. y. per kiekvieną iteraciją jis atmeta tik 38.2% intervalo, o tai akivaizdžiai mažiau nei pusė.

Taigi, apskritai kalbant apie metodų greitį mūsų tikslo funkcijai, rezultatai yra gana tikėtini: Niutono metodas yra greičiausias, aukšinio pjūvio - lėčiausias, intervalo dalijimo pusiau - tarp jų.

3.3 Efektyvumas

Optimizavimo metodų algoritmų **efektyvumas** - tikslo funkcijos iškviatimų kiekis. Kuo mažiau kartų iškviečiame funkciją tam tikro taško reikšmei apskaičiuoti, tuo mažiau išteklių sunaudojame algoritmui vykdyti, vadinasi, tuo efektyviau naudoti tam tikrą metodą. Vėlgi svarbu pažymėti, kad efektyvumas ir greitis remiasi skirtingais rodikliais (atitinkamai tikslo funkcijos iškviatimų kiekiu ir iteracijų kiekiu) ir yra visiškai nepriklausomi, todėl jei metodas yra greičiausias, jis nebūtinai yra efektyviausias, ir atvirkščiai.

Apskaičiavę kiekvienam algoritmui mūsų tikslo funkcijos (1) iškviatimų kiekį, gauname tokius rezultatus:

	Intervalo dalijimo pusiau metodas	Aukšinio pjūvio metodas	Niutono metodas
Tikslo f-jos iškviatimų skaičius	35	26	12

Atrodo, kad mūsų atveju Niutono metodas taip pat yra efektyviausias iš trijų. Tai galima paaiškinti tuo, kad mūsų pradinis spėjimas nėra labai toli nuo tikrojo minimumo taško, taip pat tuo, kad kvadratinė tikslo funkcija yra gana sklandi ir neturi tiek daug „staigių posūkių“. Tikriausiai matytume kitokį vaizdą, jei, tarkime, mūsų pradinis spėjimas būtų 100 arba funkcija būtų trigonometrinė. Tačiau kai kalbama apie kitus du metodus, matome visiškai priešingą vaizdą, nei matėme greičio kontekste. Šį kartą aukšinio pjūvio metodas yra efektyvesnis su 26 tikslo funkcijos iškviatimais, nei intervalo dalijimo pusiau su 35. Tai galima paaiškinti tuo, kad pirmasis metodas turi tik 2 taškus kaip intervalų skaičiavimus lemiančius veiksnus - x_1 ir x_2 , o antrasis - 3 taškus: x_1 , x_2 ir x_m . Aki-vaizdu, kad su kuo mažiau taškų turi dirbti metodai - tuo mažiau tikslo funkcijos iškviatimų bus atlikta, o tai atspindi bendramė kiekvieno metodo efektyvumė.

Apibendrinant, efektyviausias metodas mūsų tikslo funkcijai (1) būtų Niutono, po jo seka aukšnio pjūvio, o neefektyviausias - intervalo dalijimo pusiau.

4 Išvada

Šio darbo tikslas buvo parašyti kiekvieno iš minėtų vienmačių optimizavimo metodų algoritmą, palyginti jų rezultatus ir juos vizualizuoti, taip pat palyginti jų greitį bei efektyvumą. Baigdamas norėčiau atkreipti dėmesį į šiuos du savo pastebėjimus: pirma, kalbant apie optimizavimo metodų algoritmų greitį ir efektyvumą, jie yra visiškai nepriklausomi ir gali nekoreliuoti tarp metodų, kaip matyti iš intervalo dalijimo pusiau ir aukso pjūvio metodų pavyzdžio. Antra, kalbant konkrečiai apie Niutono metodą, nors mūsų scenarijaus atveju jis iš tiesų pasirodė greičiausias ir efektyviausias, mano nuomone, tai nereiškia, kad jis veiks taip pat ir daugeliu kitų atvejų. Jau anksčiau nustačiau, kad metodui iš tikrųjų reikia, kad pradinis spėjimas būtų kuo artimesnis minimaliam taškui, taip pat kad pati tikslo funkcija būtų kuo tolygesnė.

5 Priedas

Pilnas Python kodas:

```
1 '''
2 Matematikos ir informatikos fakultetas
3 Nikita Gainulin
4 3 kursas
5 Laboratorinis darbas 1
6 '''
7
8 import math
9 from datetime import datetime
10 import sympy as sp
11 import numpy as np
12 import matplotlib.pyplot as plt
13 print(f"{datetime.now()}")
14
15 def f(x):
16     return ((x**2-5)**2)/7-1
17
18 def halfCut(f, l, r, eps=1e-4):
19     iterNum = 0
20     funcNum = 0
21
22     xm = (l+r)/2
23     fxm = f(xm)
24     funcNum+=1
25
26     xVal = [xm]
27     yVal = [fxm]
28     while(r-l) > eps:
29         iterNum += 1
30         L = r-l
31
32         x1 = l+L/4
33         x2 = r-L/4
34
35         fx1 = f(x1)
36         fx2 = f(x2)
```

```

37
38     funcNum += 2
39
40     if fx1 < fxm:
41         r = xm
42         xm = x1
43         fxm = fx1
44         xVal.append(x1)
45         yVal.append(fx1)
46
47     elif fx2 < fxm:
48         l = xm
49         xm = x2
50         fxm = fx2
51         xVal.append(x2)
52         yVal.append(fx2)
53
54     else:
55         l = x1
56         r = x2
57         xVal.append(xm)
58         yVal.append(fxm)
59
60     xm = (l+r)/2
61     xVal.append(xm)
62     yVal.append(f(xm))
63     return xm, f(xm), iterNum, funcNum, xVal, yVal
64
65 result = halfCut(f, 0, 10)
66 print("Interval div. Minimum point:", result[0])
67 print("Interval div. Function value at minimum:", result[1])
68 print("Interval div. Iterations:", result[2])
69 print("Interval div. Functions invoked:", result[3])
70
71 xVal = np.linspace(0, 10, 100)
72 yVal = [f(x) for x in xVal]
73
74 plt.figure(figsize=(10, 6))
75 plt.plot(xVal, yVal, label='f(x) = ((x**2-5)**2)/7 - 1')
76 plt.scatter(result[4][: -1], result[5][: -1], color='red', label='Band. taškai')
77 plt.scatter(result[4][ -1], result[5][ -1], color='green', s=100, label='Min. taškas')
78 plt.xlabel('x')
79 plt.ylabel('f(x)')
80 plt.title("Intervalo dalijimo pusiau metodas")
81 plt.legend()
82 plt.grid(True)
83 plt.show()
84
85 def goldCut(f, l, r, eps=1e-4):
86     tau = (math.sqrt(5)-1)/2
87     print(tau)
88
89     L = r-l
90     x1 = l+(1-tau)*L
91     x2 = l+tau*L
92     fx1 = f(x1)
93     fx2 = f(x2)
94
95     iterNum = 0
96     funcNum = 2

```

```

97
98     xVal = [x1, x2]
99     yVal = [fx1, fx2]
100     while(r-l) > eps:
101         iterNum += 1
102         if fx1<fx2:
103             r = x2
104             x2 = x1
105             fx2 = fx1
106             L = r-l
107             x1 = l+(1-tau)*L
108             fx1 = f(x1)
109             funcNum += 1
110             xVal.append(x1)
111             yVal.append(fx1)
112         else:
113             l = x1
114             x1 = x2
115             fx1 = fx2
116             L = r-l
117             x2 = l+tau*L
118             fx2 = f(x2)
119             funcNum += 1
120             xVal.append(x2)
121             yVal.append(fx2)
122
123     xm = (l+r)/2
124     xVal.append(xm)
125     yVal.append(f(xm))
126     return xm, f(xm), iterNum, funcNum, xVal, yVal
127
128 result2 = goldCut(f, 0, 10)
129 print("Gold cut Minimum point:", result2[0])
130 print("Gold cut Function value at minimum:", result2[1])
131 print("Gold cut Iterations:", result2[2])
132 print("Gold cut Functions invoked:", result2[3])
133
134 plt.figure(figsize=(10, 6))
135 plt.plot(xVal, yVal, label='f(x) = ((x**2-5)**2)/7 - 1')
136 plt.scatter(result2[4][:-1], result2[5][:-1], color='red', label='Band. taškai')
137 plt.scatter(result2[4][-1], result2[5][-1], color='green', s=100, label='Min. taškas')
138
139 plt.xlabel('x')
140 plt.ylabel('f(x)')
141 plt.title("Auksinio pjūvio metodas")
142 plt.legend()
143 plt.grid(True)
144 plt.show()
145
146 def newton(fSym, x0, eps=1e-4, maxIter=100):
147     x = sp.symbols('x')
148     fIsv = sp.diff(fSym, x)
149     fIsv2 = sp.diff(fIsv, x)
150
151     fSk = sp.lambdify(x, fSym)
152     fIsvSk = sp.lambdify(x, fIsv)
153     fIsv2Sk = sp.lambdify(x, fIsv2)
154
155     xi = x0

```

```

156     iterNum = 0
157     funcNum = 0
158
159     xVal = [xi]
160     yVal = [fSk(xi)]
161
162     while iterNum < maxIter:
163         fIsv_xi = fIsvSk(xi)
164         fIsv2_xi = fIsv2Sk(xi)
165         funcNum += 2
166
167         if abs(fIsv2_xi) < eps:
168             break
169
170         xiNext = xi - fIsv_xi / fIsv2_xi
171         iterNum += 1
172
173         xVal.append(xiNext)
174         yVal.append(fSk(xiNext))
175
176         if abs(xiNext - xi) < eps:
177             break
178
179         xi = xiNext
180
181     return xi, fSk(xi), iterNum, funcNum, xVal, yVal
182
183 x = sp.symbols('x')
184 fSym = ((x**2 - 5)**2) / 7 - 1
185
186 x0 = 5
187 result3 = newton(fSym, x0)
188
189 print("Newton Minimum point:", result3[0])
190 print("Newton Function value at minimum:", result3[1])
191 print("Newton Iterations:", result3[2])
192 print("Newton Functions invoked:", result3[3])
193
194 xVal = np.linspace(0, 6, 100)
195 yVal = [float(fSym.subs(x, xi)) for xi in xVal]
196
197 plt.figure(figsize=(10, 6))
198 plt.plot(xVal, yVal, label='f(x) = ((x**2 - 5)**2) / 7 - 1')
199 plt.scatter(result3[4][: -1], result3[5][: -1], color='red', label='Band. taškai')
200 plt.scatter(result3[4][ -1], result3[5][ -1], color='green', s=100, label='Min. taškas')
201
202 plt.xlabel('x')
203 plt.ylabel('f(x)')
204 plt.title("Niutono metodas")
205 plt.legend()
206 plt.grid(True)
207 plt.show()

```

Tiesioginė nuoroda atsisiųsti: čia arba https://www.dropbox.com/scl/fi/kn1geotq32bo3muggn7q3/Nikita_Gainulin_lab1.py