VILNIAUS UNIVERSITETAS

MATEMATIKOS IR INFORMATIKOS FAKULTETAS

Laboratorinis darbas 1

Vienmatis optimizavimas

Nikita Gainulin

VILNIUS 2024

Turinys

1 Tikslo funkcija

Visi šioje ataskaitoje minėti metodai bus pritaikyti šiai tikslo funkcijai:

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

Mano `Python` kode aprašyta taip:

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

2 Vienmačiai optimizavimo metodai ir jų algoritmai

2.1 Intervalo dalijimo pusiau metodas

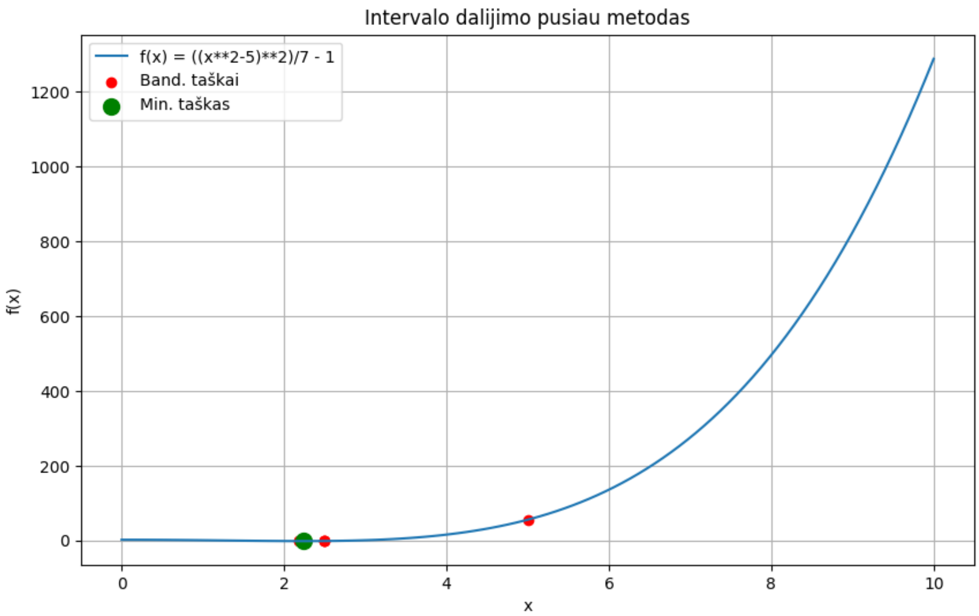
Pirmiausia reikia nustatyti intervalą, kuriame ieškosime minimumo taško. Minėto intervalo ribas įsiminsime kintamaisiais l (kairioji riba) ir r (dešinioji riba). Pradiniame intervale pasirenkam tris bandymo taškus x_1, x_2 ir x_m, taip, kad x_m būtų intervalo vidurio taškas, o x_1 ir x_2 tarp l ir vidurio 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, atitinkmai 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:

```
def halfCut(f, l, r, eps=1e-4):
    iterNum = 0
    funcNum = 0

    xm = (l+r)/2
    fxm = f(xm)
    funcNum+=1

    xVal = [xm]
    yVal = [fxm]
    while(r-l) > eps:
        iterNum += 1
        L = r-l

        x1 = l+L/4
        x2 = r-L/4

        fx1 = f(x1)
        fx2 = f(x2)

        funcNum += 2

        if fx1 < fxm:
            r = xm
            xm = x1
            fxm = fx1
            xVal.append(x1)
            yVal.append(fx1)
```

```
        elif fx2 < fxm:
            l = xm
            xm = x2
            fxm = fx2
            xVal.append(x2)
            yVal.append(fx2)

        else:
            l = x1
            r = x2
            xVal.append(xm)
            yVal.append(fxm)

    xm = (l+r)/2
    xVal.append(xm)
    yVal.append(f(xm))
    return xm, f(xm), iterNum, funcNum, xVal, yVal
```

Vizualizuojant mūsų tikslo funciją (1) šiuo metodu gauname štai tokį grafą:



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:

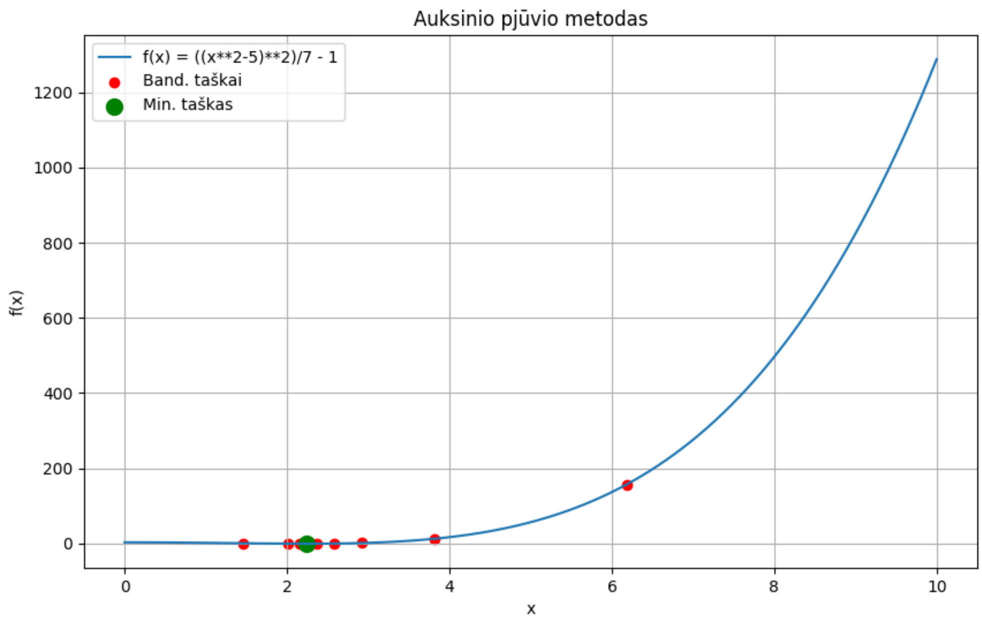
$$\tau = \frac{\sqrt{5}-1}{2} \approx 0.61803...$$

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:

```python
def goldCut(f, l, r, eps=1e-4):
    tau = (math.sqrt(5)-1)/2
    print(tau)

    L = r-l
    x1 = l+(1-tau)*L
    x2 = l+tau*L
    fx1 = f(x1)
    fx2 = f(x2)

    iterNum = 0
    funcNum = 2

    xVal = [x1, x2]
    yVal = [fx1, fx2]
    while(r-l) > eps:
        iterNum += 1
        if fx1<fx2:
            r = x2
            x2 = x1
            fx2 = fx1
            L = r-l
            x1 = l+(1-tau)*L
            fx1 = f(x1)
            funcNum += 1
            xVal.append(x1)
            yVal.append(fx1)
        else:
            l = x1
            x1 = x2
            fx1 = fx2
            L = r-l
            x2 = l+tau*L
            fx2 = f(x2)
            funcNum += 1
            xVal.append(x2)
            yVal.append(fx2)

    xm = (l+r)/2
    xVal.append(xm)
    yVal.append(f(xm))
    return xm, f(xm), iterNum, funcNum, xVal, yVal
```

Vizualizuojant mūsų tikslo funciją (1) šiuo metodu gauname štai tokį grafą:



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

2.3 Niutono metodas

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

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

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:

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

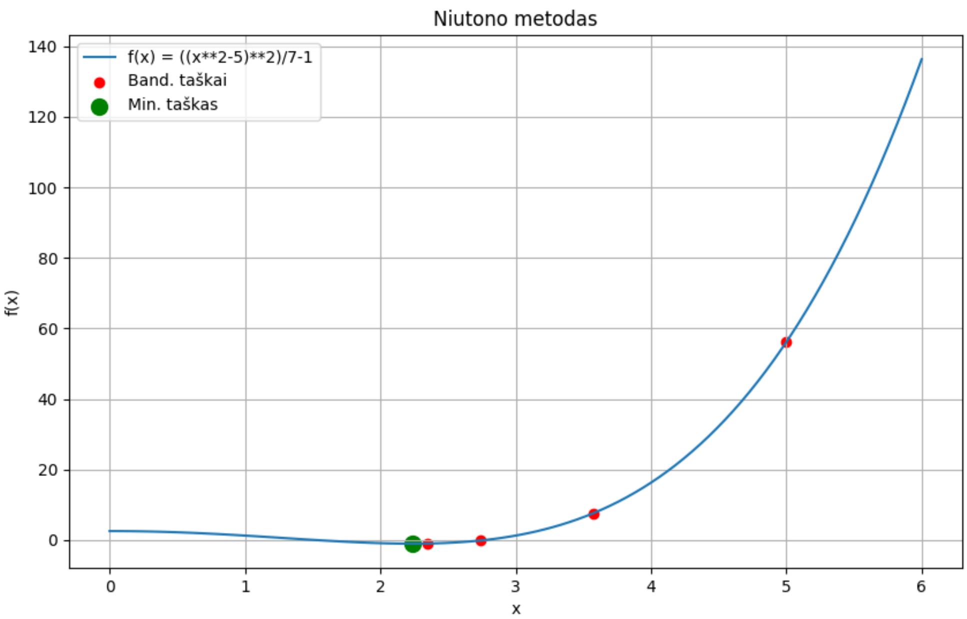
Nors ši formulė paprastai naudojama dažniau ir bendriau, formulė (2) yra tikslesnė optimizavimo kontekste, todėl ją ir naudojame vietoj šios bendrosios formulės.

Algoritmo `Python` kodas:

```
def newton(fSym, x0, eps=1e-4, maxIter=100):
    x = sp.symbols('x')

    fIsv = sp.diff(fSym, x)
    fIsv2 = sp.diff(fIsv, x)
```

```

    fSk = sp.lambdify(x, fSym)
    fIsvSk = sp.lambdify(x, fIsv)
    fIsv2Sk = sp.lambdify(x, fIsv2)

    xi = x0
    iterNum = 0
    funcNum = 0

    xVal = [xi]
    yVal = [fSk(xi)]

    while iterNum < maxIter:
        fIsv_xi = fIsvSk(xi)
        fIsv2_xi = fIsv2Sk(xi)
        funcNum += 2

        if abs(fIsv2_xi) < eps:
            break

        xiNext = xi-fIsv_xi/fIsv2_xi
        iterNum += 1

        xVal.append(xiNext)
        yVal.append(fSk(xiNext))

        if abs(xiNext-xi) < eps:
            break

        xi = xiNext

    return xi, fSk(xi), iterNum, funcNum, xVal, yVal

x = sp.symbols('x')
fSym = ((x**2-5)**2)/7-1
```

Vizualizuojant mūsų tikslo funciją (1) šiuo metodu gauname štai tokį grafą:



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	Auksinio pjūvio metodas	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	Auksinio 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 auksinio 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 auksinio pjūvio metodas perkelia 61.8% ankstesnės iteracijos intervalo (nes $\tau \approx 0.61803...$), 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, auksinio pjūvio - lėčiausias, intervalo dalijimo pusiau - tarp jų.

3.3 Efektyvumas

Optimizavimo metodų algoritmų **efektyvumas** - tikslo funkcijos iškvietimų 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 iškvietimų 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) iškvietimų kiekį, gauname tokius rezultatus:

	Intervalo dalijimo pusiau metodas	Auksinio pjūvio metodas	Niutono metodas
Tikslo f-jos iškvietimų 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 auksinio pjūvio metodas yra efektyvesnis su 26 tikslo funkcijos iškvietimais, nei intervalo dalijimo pusiau su 35. Tai galima paaiškinti tuo, kad pirmasis metodas turi tik 2 taškus kaip intervalų skaičiavimus lemiančius veiksnius - x_1 ir x_2, o antrasis - 3 taškus: x_1, x_2 ir x_m. Akivaizdu, kad su kuo mažiau taškų turi dirbti metodai - tuo mažiau tikslo funkcijos iškvietimų bus atlikta, o tai atsispindi bendrame kiekvieno metodo efektyvume.

Apibendrinant, efektyviausias metodas mūsų tikslo funkcijai (1) būtų Niutono, po jo seka auksinio 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:

```
'''
Matematikos ir informatikos fakultetas
Nikita Gainulin
3 kursas
Laboratorinis darbas 1
'''

import math
from datetime import datetime
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
print(f"{datetime.now()}")

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

def halfCut(f, l, r, eps=1e-4):
    iterNum = 0
    funcNum = 0

    xm = (l+r)/2
    fxm = f(xm)
    funcNum+=1

    xVal = [xm]
    yVal = [fxm]
    while(r-l) > eps:
        iterNum += 1
        L = r-l

        x1 = l+L/4
        x2 = r-L/4

        fx1 = f(x1)
        fx2 = f(x2)
```

```

        funcNum += 2

        if fx1 < fxm:
            r = xm
            xm = x1
            fxm = fx1
            xVal.append(x1)
            yVal.append(fx1)

        elif fx2 < fxm:
            l = xm
            xm = x2
            fxm = fx2
            xVal.append(x2)
            yVal.append(fx2)

        else:
            l = x1
            r = x2
            xVal.append(xm)
            yVal.append(fxm)

    xm = (l+r)/2
    xVal.append(xm)
    yVal.append(f(xm))
    return xm, f(xm), iterNum, funcNum, xVal, yVal

result = halfCut(f, 0, 10)
print("Interval div. Minimum point:", result[0])
print("Interval div. Function value at minimum:", result[1])
print("Interval div. Iterations:", result[2])
print("Interval div. Functions invoked:", result[3])

xVal = np.linspace(0, 10, 100)
yVal = [f(x) for x in xVal]

plt.figure(figsize=(10, 6))
plt.plot(xVal, yVal, label='f(x) = ((x**2-5)**2)/7 - 1')
plt.scatter(result[4][:-1], result[5][:-1], color='red', label='Band. taškai')
plt.scatter(result[4][-1], result[5][-1], color='green', s=100, label='Min. taškas')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title("Intervalo dalijimo pusiau metodas")
plt.legend()
plt.grid(True)
plt.show()

def goldCut(f, l, r, eps=1e-4):
    tau = (math.sqrt(5)-1)/2
    print(tau)

    L = r-l
    x1 = l+(1-tau)*L
    x2 = l+tau*L
    fx1 = f(x1)
    fx2 = f(x2)

    iterNum = 0
    funcNum = 2
```

```

    xVal = [x1, x2]
    yVal = [fx1, fx2]
    while(r-l) > eps:
        iterNum += 1
        if fx1<fx2:
            r = x2
            x2 = x1
            fx2 = fx1
            L = r-l
            x1 = l+(1-tau)*L
            fx1 = f(x1)
            funcNum += 1
            xVal.append(x1)
            yVal.append(fx1)
        else:
            l = x1
            x1 = x2
            fx1 = fx2
            L = r-l
            x2 = l+tau*L
            fx2 = f(x2)
            funcNum += 1
            xVal.append(x2)
            yVal.append(fx2)

    xm = (l+r)/2
    xVal.append(xm)
    yVal.append(f(xm))
    return xm, f(xm), iterNum, funcNum, xVal, yVal

result2 = goldCut(f, 0, 10)
print("Gold cut Minimum point:", result2[0])
print("Gold cut Function value at minimum:", result2[1])
print("Gold cut Iterations:", result2[2])
print("Gold cut Functions invoked:", result2[3])

plt.figure(figsize=(10, 6))
plt.plot(xVal, yVal, label='f(x) = ((x**2-5)**2)/7 - 1')
plt.scatter(result2[4][:-1], result2[5][:-1], color='red', label='Band. taškai')
plt.scatter(result2[4][-1], result2[5][-1], color='green', s=100, label='Min. taškas')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title("Auksinio pjūvio metodas")
plt.legend()
plt.grid(True)
plt.show()

def newton(fSym, x0, eps=1e-4, maxIter=100):
    x = sp.symbols('x')

    fIsv = sp.diff(fSym, x)
    fIsv2 = sp.diff(fIsv, x)

    fSk = sp.lambdify(x, fSym)
    fIsvSk = sp.lambdify(x, fIsv)
    fIsv2Sk = sp.lambdify(x, fIsv2)

    xi = x0
```

```
    iterNum = 0
    funcNum = 0

    xVal = [xi]
    yVal = [fSk(xi)]

    while iterNum < maxIter:
        fIsv_xi = fIsvSk(xi)
        fIsv2_xi = fIsv2Sk(xi)
        funcNum += 2

        if abs(fIsv2_xi) < eps:
            break

        xiNext = xi-fIsv_xi/fIsv2_xi
        iterNum += 1

        xVal.append(xiNext)
        yVal.append(fSk(xiNext))

        if abs(xiNext-xi) < eps:
            break

        xi = xiNext

    return xi, fSk(xi), iterNum, funcNum, xVal, yVal

x = sp.symbols('x')
fSym = ((x**2-5)**2)/7-1

x0 = 5
result3 = newton(fSym, x0)

print("Newton Minimum point:", result3[0])
print("Newton Function value at minimum:", result3[1])
print("Newton Iterations:", result3[2])
print("Newton Functions invoked:", result3[3])

xVal = np.linspace(0, 6, 100)
yVal = [float(fSym.subs(x, xi)) for xi in xVal]

plt.figure(figsize=(10,6))
plt.plot(xVal, yVal, label='f(x) = ((x**2-5)**2)/7-1')
plt.scatter(result3[4][:-1], result3[5][:-1], color='red', label='Band. taškai')
plt.scatter(result3[4][-1], result3[5][-1], color='green', s=100, label='Min. taškas'
    )
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title("Niutono metodas")
plt.legend()
plt.grid(True)
plt.show()
```

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