VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS

Laboratorinis darbas 2

Optimizavimas be apribojimų

Nikita Gainulin

VILNIUS 2024

Turinys

1 Įvadas

Savo ankstesniame laboratoriniame darbe gilinausi į vienmatį optimizavimą. Kaip nustačiau, daugu-ma šių algoritmų remiasi tuo, kad naudotojas turi pasirinkti konkretų intervalą, kuriame bus ieškoma minimumo taško. Tačiau šį kartą nagrinėsiu optimizavimą be apribojimų, kurio algoritmams, kaip galima spėti iš pavadinimo, nebūtinai reikia iš anksto nustatyto intervalo, nes jų leistinoji sritis sutampa su visa n-mate Euklido erdve $\mathbb{R}^n$ ir jiems veikti reikės kitokio kintamųjų rinkinio. Be to, šiame laboratoriniame darbe taip pat bandysiu optimizuoti konkrečią problemą, panašią į tas, kurios iš tikrųjų pasitaiko realiame pasaulyje.

2 Nagrinėjama problema

Prieš tęsdami nustatysime tikrąją problemą, kurią bandysiu optimizuoti. Štai kaip ji skamba:

Kokia turėtų būti stačiakampio gretasienio formos dėžė, kad vienetiniam paviršiaus plotui jos tūris būtų maksimalus?

Pirmiausia, atsižvelgiant į šio laboratorinio darbo temą, tikslo funkciją būtina aprašyti taip, kad pats optimizavimo uždavinys būtų sudarytas be apribojimų, t. y. $\min f(X)$. Kaip jau žinome, standartinė tūrio formulė yra tokia:

$$V = a \cdot b \cdot c,$$

kur a, b ir c yra mūsų stačiakampio gretasienio ilgis, plotis ir aukštis. Tačiau dėl paprastumo dirbsime su tūrio kvadratu, nes vėliau atlikdami pakeitimą gausime daugianarę išraišką, todėl bus daug lengviau rasti išvestines ir kritinius taškus. Apibrėžkime:

- $x_1 = 2ab$, priekinės ir galinės sienų plotų suma;
- $x_2 = 2bc$, šoninių sienų plotų suma;
- $x_3 = 2ac$, viršutinės ir apatinės sienų plotų suma;

Iš čia:

- $ab = \frac{x_1}{2}$;
- $bc = \frac{x_2}{2}$;
- $ac = \frac{x_3}{2}$;

Kadangi mūsų užduotis yra maksimaliai padidinti dėžės tūrį, tenkantį vienam paviršiaus ploto vienetui, mūsų reikalavimas tampa $x_1 + x_2 + x_3 = 1$. Čia galime išreikšti $x_3 = 1 - x_1 - x_2$.Taip pradinį optimizavimo uždavinį transformuojame į neapribotą optimizavimo uždavinį. Kadangi mus domina tik tūrio kvadratas, toliau atliekami tokie veiksmai:

$$V^2 = (abc)^2 = aabbcc = \frac{x_1}{2} \cdot \frac{x_2}{2} \cdot \frac{x_3}{2} = \frac{1}{8} \cdot x_1 x_2 x_3 = \frac{1}{8} \cdot x_1 x_2 \cdot (1 - x_1 - x_2) = \frac{1}{8} \cdot (x_1 x_2 - x_1^2 x_2 - x_1 x_2^2) \quad (1)$$

Kadangi mūsų tikslas yra optimizuoti uždavinį ieškant minimumo taško, kuriame dėžės tūris yra didžiausias, turime padauginti 1 formulę iš -1, nes didžiausia V^2 vertė atitinka mažiausią $-V^2$. Štai tai ir gauname mūsų objektinę funkciją:

$$f(x) = -\frac{1}{8}(x_1 x_2 - x_1^2 x_2 - x_1 x_2^2) \quad (2)$$

2.1 Objektinė funkcija ir jos gradientas

Ankstesniame skyriuje nustatėme tokią optimizavimo uždavinio tikslo funkciją (2):

$$f(x) = -\frac{1}{8}(x_1x_2 - x_1^2x_2 - x_1x_2^2)$$

Dviem iš trijų algoritmų, kuriuos nagrinėsiu šiame laboratoriniame darbe, reikalingas vadinamasis tikslo funkcijos gradientas. **Gradientas** - vektorius, sudarytas iš funkcijos dalinių išvestinių, apskaičiuotų taške x.

$$\nabla f(x) = (\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_n})$$

Mūsų tikslinei funkcijai tai gana paprasta - ją tereikia diferencijuoti pagal x_1 ir x_2 atskirai:

$$\frac{\partial f(x)}{\partial x_1} = -\frac{1}{8}(x_2 - 2x_1x_2 - x_2^2)$$

$$\frac{\partial f(x)}{\partial x_2} = -\frac{1}{8}(x_1 - 2x_1x_2 - x_1^2)$$

Taigi, gavome savo tikslo funkcijos gradientą:

$$\nabla f(x) = (-\frac{1}{8}(x_2 - 2x_1x_2 - x_2^2), -\frac{1}{8}(x_1 - 2x_1x_2 - x_1^2)) \tag{3}$$

Savo patogumui sukūriau tikslo funkcijos ir jos gradiento Python klasę, nes ji leidžia man įdiegti vidinį skaitiklį, kuris leidžia daug tiksliau apskaičiuoti visą funkcijos iškvietimą, taip pat funkciją, kuri jį atstato. Taip man nebereikės gaišti laiko ieškant vietos kode, kur rankiniu būdu padidinti skaitiklį - tai buvo vienas iš mano pirmojo laboratorinio darbo aplaidumų.

```
class ObjectiveFunction:
    counter = 0

    def __init__(self):
        pass

    def f(self, x):
        ObjectiveFunction.counter += 1
        return -1/8*(x[0]*x[1]-x[0]**2*x[1]-x[0]*x[1]**2)

    def gradF(self, x):
        ObjectiveFunction.counter += 1
        return [-1/8*(x[1]-2*x[0]*x[1]-x[1]**2), -1/8*(x[0]-2*x[0]*x[1]-x[0]**2)]

    def reset(self):
        ObjectiveFunction.counter = 0
```

3 Optimizavimo be apribojimų metodai ir jų algoritmai

Šiame skyriuje papasakosiu apie pačius metodus, kaip veikia jų algoritmai, ir pateiksiu savo asmeninę kiekvieno algoritmo interpretaciją.

3.1 Gradientinio nusileidimo metodas

Paprastai nusileidimo metodai pagrįsti informacija apie tikslo funkcijos $f(x)$ pirmąją ir antrąją dalines išvestines. Pats gradientas nukreiptas funkcijos greičiausio augimo kryptimi, todėl, norėdami rasti minimumo tašką abiem nusileidimo metodais, žengsime būtent antigradiento kryptimi. Tai ir yra esminė priežastis kodėl mūsų objektinės funkcijos (2) priekyje yra minuso ženklas.

Panagrinėkime gradientinio nusileidimo metodą. Toliau pateikta formulė laikoma jo bei sekančio nusileidimo metodo esme:

$$x_{i+1} = x_i - \gamma \nabla f(x_i), \tag{4}$$

kur x_i ir x_{i+1} - atitinkamai vienos ir sekančios iteracijų bandomieji taškai, γ - žingsnio daugiklis bei $\nabla f(x_i)$ - objektinės funkcijos gradientas taške x_i. Pradėję nuo nurodyto pradinio taško x_0, taikydami pirmiau pateiktą formulę, mūsų x_i iteracija po iteracijos artėja prie minimumo taško. Svarbu pažymėti, kad nors mūsų žingsnio daugiklis γ tiesiogiai niekada nesikeičia, artėjimo greitis ($-\gamma \nabla f(x_i)$) nėra pastovus ir kiekvieną iteraciją tampa vis mažesnis dėl to, kad mažėja gradiento norma.

Tyrinėdamas ir bandydamas parašyti Python gradiento nusileidimo algoritmo realizaciją, pastebėjau įdomų dalyką - dauguma tyrėjų mėgsta naudoti šiuos šešis žingsnio daugiklius γ: 0.001, 0.003, 0.01, 0.03, 0.1 ir 0.3. Pirmieji du man pasirodė gana maži, todėl konkrečiai šiam metodui nusprendžiau nustatyti ne 200, o 1000 iteracijų ribą, nes teoriškai net ir tokio kiekio gali nepakakti, kad su tam tikrais koeficientais būtų pasiektas minimumo taškas. Taigi, štai kaip atrodo mano gradientinio nusileidimo algoritmas:

```
from imports import np
from objfunc import ObjectiveFunction

def f(x):
    return -1/8*(x[0]*x[1]-x[0]**2*x[1]-x[0]*x[1]**2)

def gradDescent(obj:ObjectiveFunction, x0, gamma, eps=1e-4, maxIter=10000):
    iter = 0
    x = x0.copy()
    points = [x0.copy()]

    while iter < maxIter:
        grad = obj.gradF(x)
        if np.linalg.norm(grad) < eps:
            break

        x = [x[j] - gamma * grad[j] for j in range(len(x))]
        points.append(x.copy())

        '''if iter%50==0:
            print(f"[{iter}]: x = {x}, f(x) = {f(x)}, func. calls = {obj.counter}")''
                                                                    '


        iter += 1

    fX = obj.f(x)

    return x, fX, iter, points
```

3.2 Greičiausio nusileidimo metodas

Apskritai stačiausio nusileidimo metodas yra labai panašus į gradiento nusileidimo - taip pat artėja prie minimumo taško iteruojant per formulę (4), tačiau šį kartą pats algoritmas, naudodamas išorinį, nustato optimaliausią žingsnio daliklį per iteraciją. Realiai bet kuris vienmačio optimizavimo algoritmas turėtų tikti. Savo atveju nusprendžiau naudoti auksinio pjūvio algoritmą iš pirmojo laboratorinio darbo, nes man jį daug lengviau įgyvendinti. Tačiau galima teigti, kad taip elgdamasis atmetu neribotąją nusileidimo algoritmų savybę dėl to, kad auksinio pjūvio algoritmas reikalauja intervalo, tačiau tai netiesa, nes paieška tik padeda rasti efektyviausią žingsnio dydį nusileidimo kryptimi ir neriboja, kur greičiausio nusileidimo algoritmas gali eiti bendroje objektinės funkcijos erdvėje.

Taigi, aukso pjūvio intervalai, kuriuos naudosiu, yra standartinis [0,1] ir tie, kuriuos radau bandymų būdu - [0,7] ir [0,20]. Intervalų pasirinkimą plačiau aptarsiu, kai pereisime prie algoritmo rezultatų. Štai Python įgyvendinimas:

```
from objfunc import ObjectiveFunction
from imports import np
import math

def f(x):
    return -1/8*(x[0]*x[1]-x[0]**2*x[1]-x[0]*x[1]**2)

def steepDescent(obj:ObjectiveFunction, x0, eps=1e-4, maxIter=200):
    iter = 0
    x = x0.copy()
    points = [x0.copy()]
    optimalGamma = 0
    while iter < maxIter:
        grad = obj.gradF(x)

        if np.linalg.norm(grad) < eps:
            break

        def objective(gamma):
            return obj.f([x[j] - gamma * grad[j] for j in range(len(x))])

        l, r = 0, 1    # Vėliau taip pat bus nagrinėjami l,r = 0,7 bei l,r = 0,20
        tau = (math.sqrt(5)-1)/2
        L = r-l
        x1 = l+(1-tau)*L
        x2 = l+tau*L
        fx1 = objective(x1)
        fx2 = objective(x2)

        while (r-l) > eps:
            if fx1 < fx2:
                r = x2
                x2 = x1
                fx2 = fx1
                L = r-l
                x1 = l+(1-tau)*L
                fx1 = objective(x1)
            else:
                l = x1
                x1 = x2
                fx1 = fx2
                L = r-l
                x2 = l+tau*L
                fx2 = objective(x2)
```

```
        gammaOpt = (l+r)/2
        optimalGamma = gammaOpt

        x = [x[j]-gammaOpt*grad[j] for j in range(len(x))]
        points.append(x.copy())

        '''if iter%10==0:
            print(f"[{iter}]: x = {x}, f(x) = {f(x)}, func. calls = {obj.counter}")''
                                                                    '


        iter += 1

    print(optimalGamma)
    return x, obj.f(x), iter, points
```

3.3 Deformuojamo simplekso (Nelder-Mead) metodas

Skirtingai nei ankstesni du metodai, Nelderio-Medo, arba kaip jis paprastai vadinamas deformuojamo simplekso, nesiremia tikslo funkcijos gradientu, o naudoja figūras, sudarytas iš taškų, kurie iteracijų metu palaipsniui artėja prie minimumo taško. Figūra priklauso nuo paieškos erdvės matmens, kuris nustatomas pagal tai, kiek kintamųjų bandome optimizuoti, tai yra n - įvesto x_0 ilgio, ir griežtai vadovaujasi $n + 1$ formule. Mūsų atveju tai bus trikampis. Taigi algoritmui nustačius pradinį trikampį, jo taškai išrikiuojami nuo geriausio, tai yra arčiausiai minimumo taško pagal tikslo funkcijos vertę, iki blogiausio. Tada sukuriamas dar vienas taškas, vadinamas centroidu - visų taškų, išskyrus blogiausią, centro taškas. Po to algoritmas atspindi blogiausią tašką per centrą, tarsi jį apverčia, ir tada jį išsaugo arba, jei tas naujas taškas yra arčiau minimumo, dar labiau išplečia. Tačiau jei atspindėtas taškas yra toliau nei antras blogiausias - jis perkeliamas arčiau centroido. Ir šie veiksmai su figūromis tęsiasi kiekvieną iteraciją, kol taškai pakankamai priartėja vienas prie kito ir atitinkamai prie minimumo taško arba algoritmas išnaudoja leistiną iteracijų skaičių.

Taip atrodo mano deformuojamo simplekso įgyvendinimas Python kalba:

```
from objfunc import ObjectiveFunction

def f(x):
    return -1/8*(x[0]*x[1]-x[0]**2*x[1]-x[0]*x[1]**2)

def simplex(obj:ObjectiveFunction, x0, initStep=0.025, stepCoeff=1.025, eps=1e-4,
                                            alpha=1, gamma=2, rho=0.5, maxIter=200):
    n = len(x0)
    pond = [x0]
    fVals = [obj.f(x0)]
    it = 0

    triangles = []

    if n == 2:
        pond.append([x0[0] + initStep, x0[1]])
        fVals.append(obj.f(pond[-1]))

        pond.append([x0[0] + initStep/2, x0[1] + initStep * 0.866])
        fVals.append(obj.f(pond[-1]))
    else:
        for i in range(n):
            xTemp = x0.copy()
```

```
            if xTemp[i] == 0:
                xTemp[i] = initStep
            else:
                xTemp[i] *= stepCoeff
            pond.append(xTemp)
            fVals.append(obj.f(xTemp))

    triangles.append(pond.copy())

    while True and it < maxIter:
        it += 1

        sortedInd = sorted(range(len(fVals)), key=lambda i: fVals[i])
        pond = [pond[i] for i in sortedInd]
        fVals = [fVals[i] for i in sortedInd]

        triangles.append(pond.copy())

        x_m = [sum([pond[i][j] for i in range(len(pond) - 1)]) / n for j in range(n)]

        x_r = [x_m[j] + alpha * (x_m[j] - pond[-1][j]) for j in range(n)]
        y_r = obj.f(x_r)

        if fVals[0] <= y_r < fVals[-2]:
            pond[-1], fVals[-1] = x_r, y_r
        elif y_r < fVals[0]:
            x_e = [x_m[j] + gamma * (x_m[j] - pond[-1][j]) for j in range(n)]
            y_e = obj.f(x_e)
            if y_e < y_r:
                pond[-1], fVals[-1] = x_e, y_e
            else:
                pond[-1], fVals[-1] = x_r, y_r
        else:
            x_c = [x_m[j] + rho * (pond[-1][j] - x_m[j]) for j in range(n)]
            y_c = obj.f(x_c)
            if y_c < fVals[-1]:
                pond[-1], fVals[-1] = x_c, y_c

        xErr = max([sum((pond[i][j] - pond[0][j]) ** 2 for j in range(n)) ** 0.5 for
                                                    i in range(1, len(pond))])
        yErr = abs(fVals[0] - fVals[-1])

        if xErr < eps and yErr < eps:
            break

        #print(it)
        if (it-1)%5==0:
            print(f"[{it-1}]: x = {pond[0]}, f(x) = {f(pond[0])}, func. calls = {obj.
                                                    counter}")

    return pond[0], fVals[0], it, triangles
```

4 Rezultatai ir jų analyzė

Prieš pradedant rezultatų analizę, svarbu nustatyti kelias pagrindines sąvokas, kurias naudojau ir ankstesniame laboratoriniame darbe, taip pat pradinius taškus, kuriuos nagrinėsiu.

Optimizavimo metodo algoritmo **greitis** - tai jo vidinių iteracijų kiekis, per kurį randamas tikslus

minimumo taškas (arba intervalas, kuriame yra tas taškas ir kuris yra mažesnis už iš anksto nustatytą ε). Paprastai kuo mažiau iteracijų algoritmui reikia minėtam minimumo taškui pasiekti, tuo jis yra greitesnis.

Optimizavimo metodo algoritmo **efektyvumas** - tikslo funkcijos bei jos gradiento iškvietimų kiekis. Kuo mažiau kartų iškviečiame funkciją arba jos gradientą tam tikro taško reikšmei apskaičiuoti, tuo mažiau išteklių sunaudojame algoritmui vykdyti, vadinasi, tuo efektyviau naudoti tam tikrą metodą.

Trys pradiniai taškai x_0, kuriuos įvesiu į algoritmus, yra šie: [0,0], [1,1] ir [0.5,0.7].

Dabar, kai pagrindinės sąvokos ir nuliniai taškai apibrėžti, pereikime prie analizės.

4.1 Gradientinio nusileidimo metodo rezultatai ir vizualizacija

Taikydamas šį metodą galėjau laisvai nustatyti žingsnio daugiklį γ taip, kaip man atrodė tinkama. Ištyręs daugybę internetinių šaltinių, kuriuose nagrinėjamas gradientinis nusileidimas, nustačiau, kad populiariausi buvo šie šeši: 0.001, 0.003, 0.01, 0.03, 0.1 ir 0.3. Kaip jau minėjau, žiūrėdamas į šias γ jau abejojau, kad kai kurios iš jų pasieks minimumo tašką, todėl šiam algoritmui nusprendžiau padidinti didžiausią iteracijų skaičių iki 1000, tačiau, kaip pamatysime, net ir to nepakako.

4.1.1 Minimumo taškas ir funkcijos reikšmė

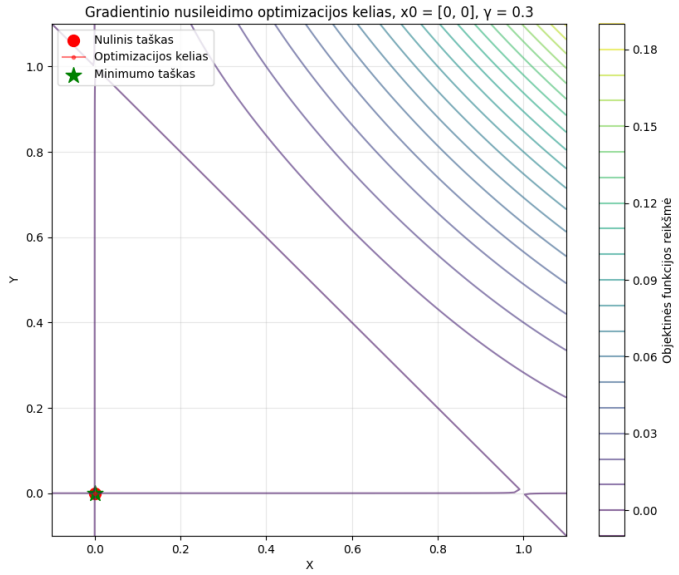
Viską apskaičiavęs, gavau šiuos minimumo taškus ir funkcijų reikšmes, skirtas anksčiau minėtiems trims pradiniams taškams ir šešioms γ:

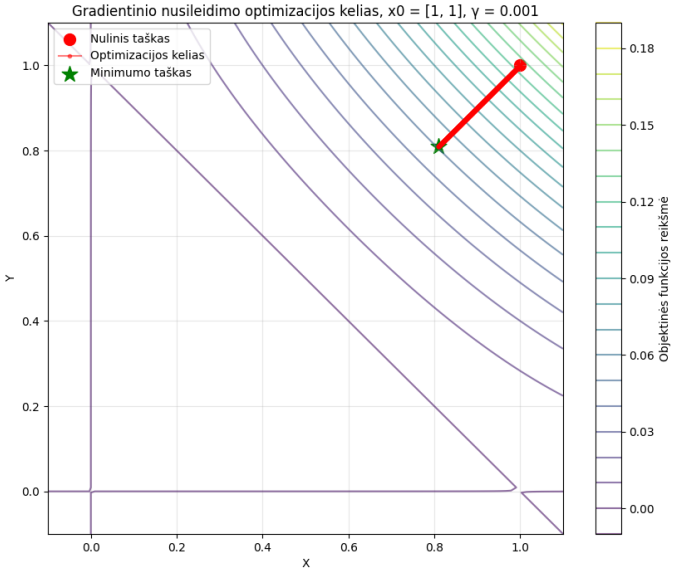
		Nuliniai taškai x_0					
		[0,0]		[1,1]		[0.5,0.7]	
		Min. taškas	Funkc. reikšmė	Min. taškas	Funkc. reikšmė	Min. taškas	Funkc. reikšmė
Žingsnio daugikliai γ	0.001	[0, 0]	0	[0.809672, 0.809672]	0.050753	[0.447687, 0.651397]	0.003611
	0.01	[0, 0]	0	[0.411849, 0.411849]	-0.003738	[0.297361, 0.477088]	-0.003999
	0.1	[0, 0]	0	[0.333891, 0.333891]	-0.004629	[0.330972, 0.335728]	-0.004629
	0.003	[0, 0]	0	[0.615093, 0.615093]	0.010886	[0.380095, 0.585160]	-0.000965
	0.03	[0, 0]	0	[0.338583, 0.338583]	-0.004626	[0.296528, 0.383428]	-0.004548
	0.3	[0, 0]	0	[0.333881, 0.333881]	-0.004629	[0.331648, 0.335034]	-0.004629

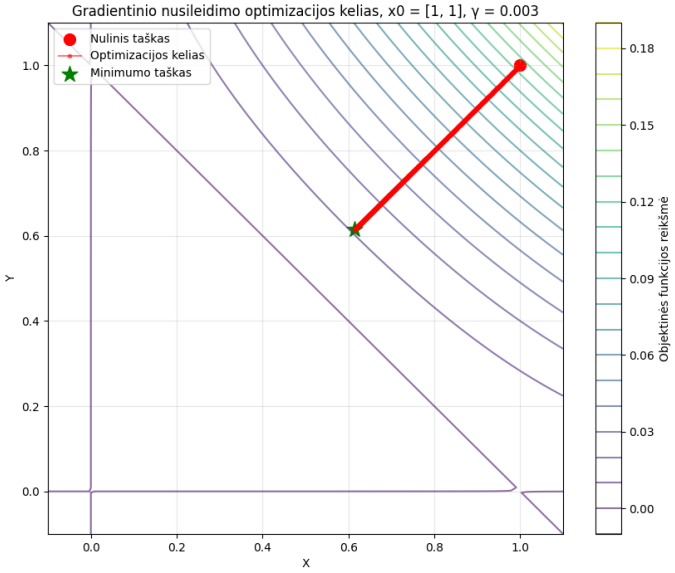
1 lentelė: Gradientinio nusileidimo metodo minimumo taškai ir funkcijos reikšmės visiems γ ir x_0

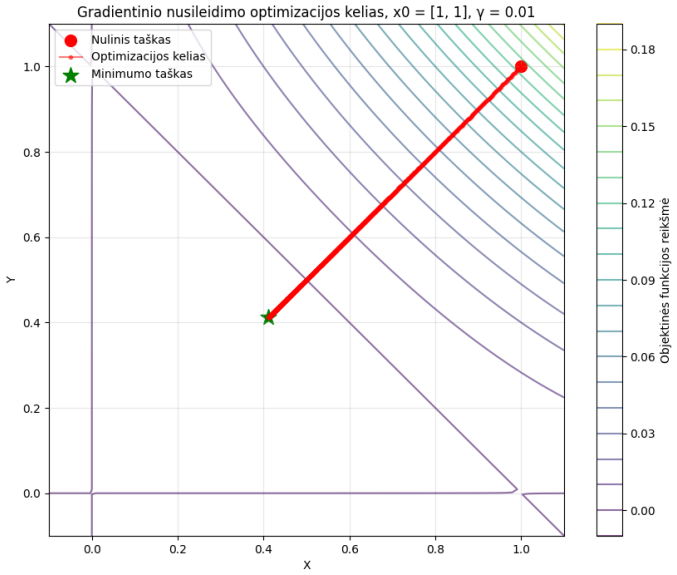
Kaip matome, taške [0,0] ir tikslo funkcija, ir jos gradientas yra nuliniai. Tai gana akivaizdu, nes šio taško gradiento norma visada bus lygi 0, t. y. mažesnė už numatytąją ε reikšmę, todėl algoritmas net negali užbaigti pirmosios iteracijos ir yra priverstas grąžinti tašką [0,0]. Kalbant apie kitus taškus, atrodo, kad buvau teisus - kai kurios pateiktos γ reikšmės iš tiesų yra per mažos, kad algoritmas galėtų pasiekti norimą minimumo tašką net po 1000 iteracijų. Tai atsispindi tiek [1,1], tiek [0,5,0,7] pradinių taškų rezultatuose: 0.001, 0.003, 0.01 ir 0.03 yra tiesiog per maži žingsnio daugikliai šiam konkrečiam metodui. Gali kilti akivaizdus klausimas: „Gerai, o kuriuos tada mums bus geriausia naudoti?“. Tačiau tai ir yra kito metodo esmė, o į patį klausimą bus atsakyta jo analizės metu. Kalbant apie kitas dvi γ: jų funkcijų reikšmės yra beveik identiškos, o minimumo taškai tik šiek tiek skiriasi. Tačiau skirtumas tarp jų yra nereikšmingas, todėl abu atsakymai laikomi teisingais.

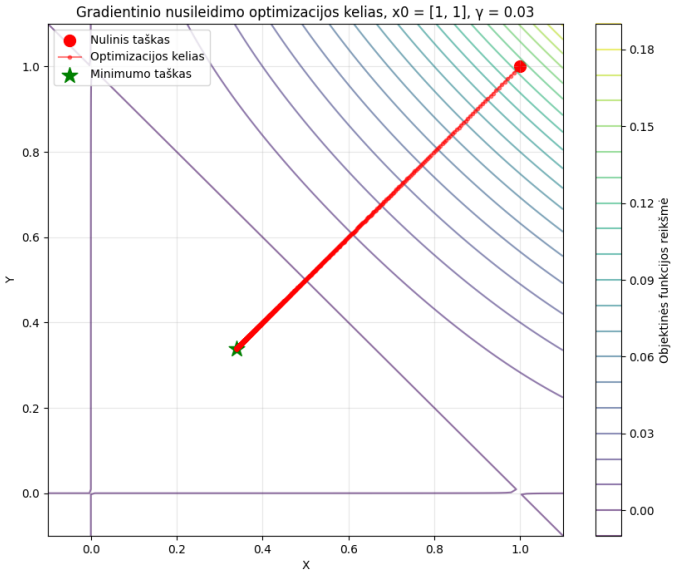
Taip atrodo rezultatų vizualizacija kontūriniuose grafikuose:

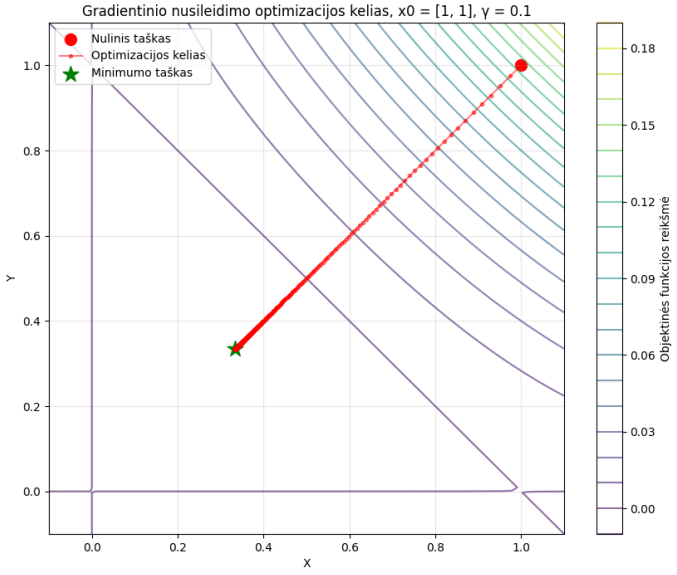

Gradientinio nusileidimo optimizacijos kelias, x0 = [0, 0], γ = 0.3
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė

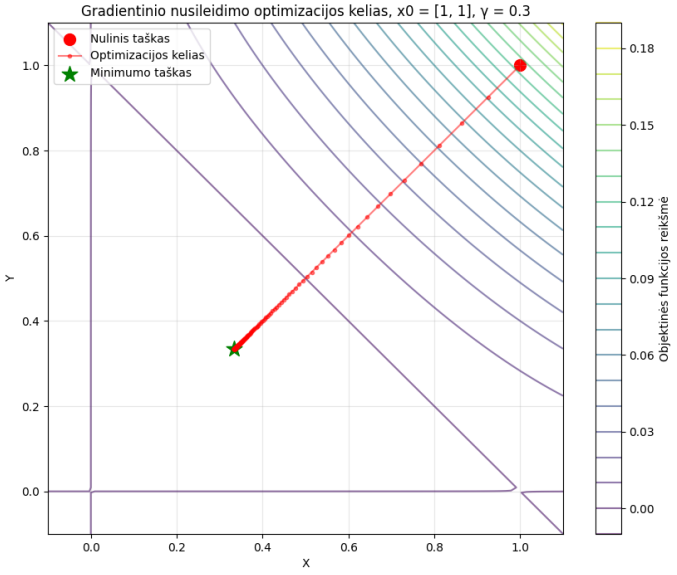

Gradientinio nusileidimo optimizacijos kelias, x0 = [1, 1], γ = 0.001
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė

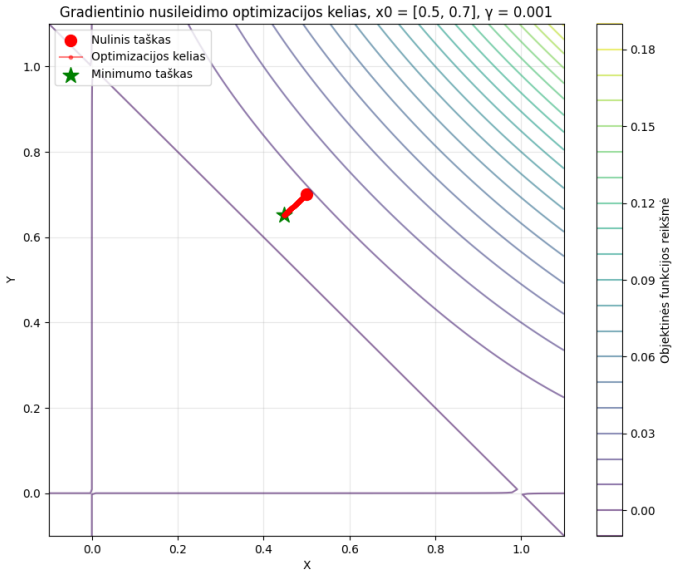

Gradientinio nusileidimo optimizacijos kelias, x0 = [1, 1], γ = 0.003
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė

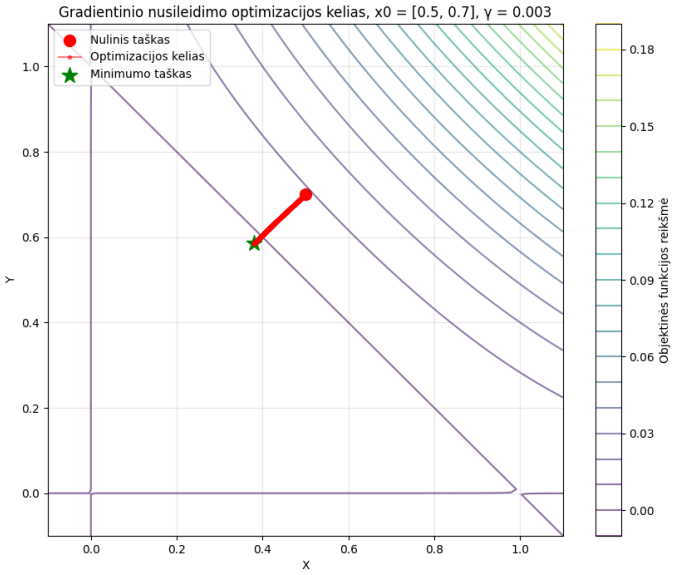

Gradientinio nusileidimo optimizacijos kelias, x0 = [1, 1], γ = 0.01
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė

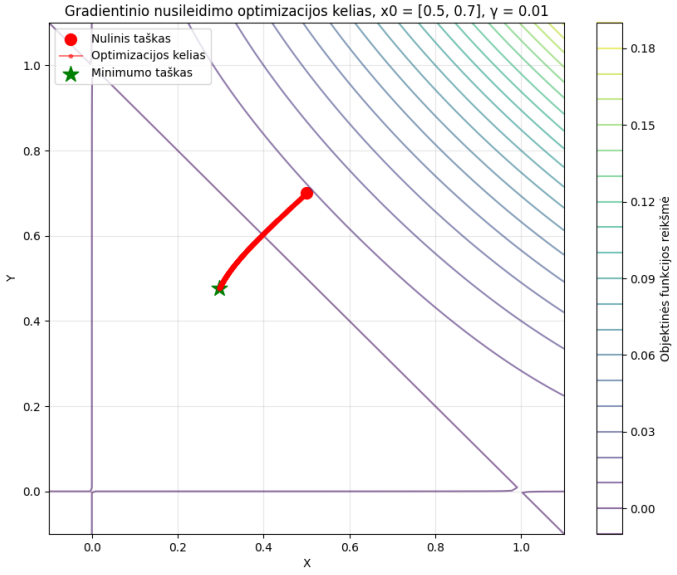

Gradientinio nusileidimo optimizacijos kelias, x0 = [1, 1], γ = 0.03
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė

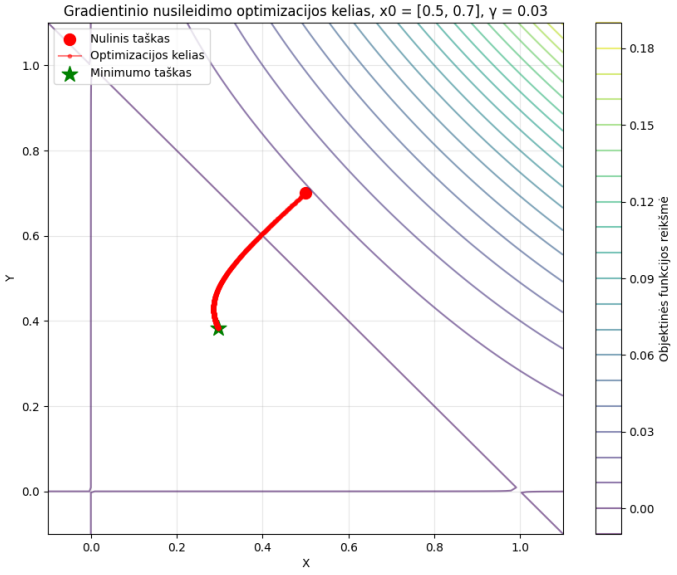

Gradientinio nusileidimo optimizacijos kelias, x0 = [1, 1], γ = 0.1
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė

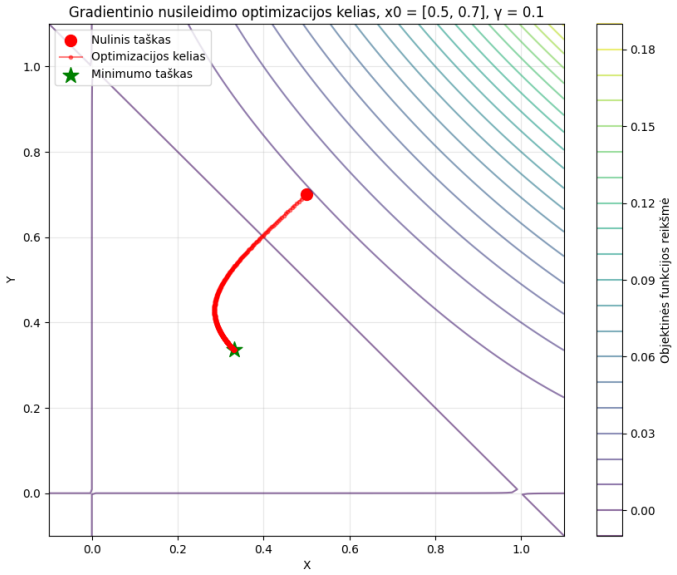

Gradientinio nusileidimo optimizacijos kelias, x0 = [1, 1], γ = 0.3
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė

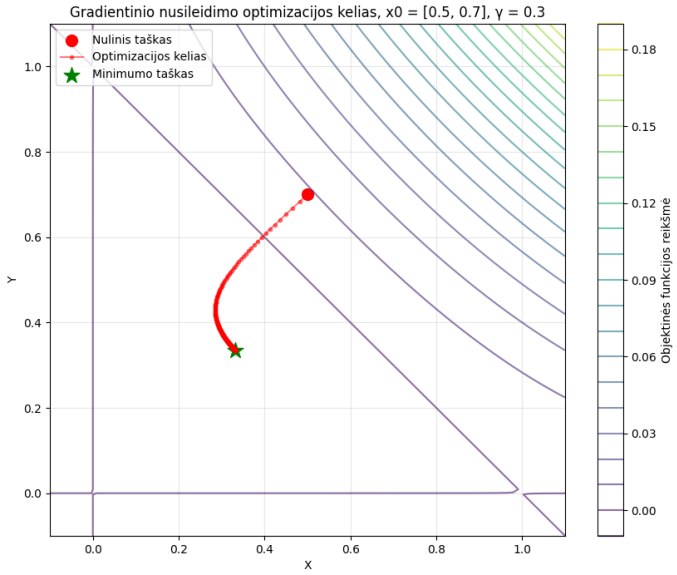

Gradientinio nusileidimo optimizacijos kelias, x0 = [0.5, 0.7], γ = 0.001
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė


Gradientinio nusileidimo optimizacijos kelias, x0 = [0.5, 0.7], γ = 0.003
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė


Gradientinio nusileidimo optimizacijos kelias, x0 = [0.5, 0.7], γ = 0.01
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė


Gradientinio nusileidimo optimizacijos kelias, x0 = [0.5, 0.7], γ = 0.03
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė





Kaip matyti iš grafikų, žingsnio daugikliai γ daugiausia lemia taškų ir iteracijų, kurias algoritmas turi atlikti, kol pasiekia minimumo tašką, kiekį. Kuo storesnė linija, jungianti pradinį tašką su surastu minimumu, tuo daugiau tarpinių taškų algoritmas turėjo pereiti, taigi tuo daugiau ir iteracijų, ir funkcijos iškvietimų. Be to, kaip minėjau anksčiau, artėjimo greitis ($-\gamma\nabla f(x_i)$) niekada nebūna pastovus, ir tai matyti iš grafikų didesnėms γ reikšmėms - 0.1 ir 0.3.

4.1.2 Greitis

Čia yra gradientinio nusileidimo metodo greičio lentelė:

		Nuliniai taškai x_0		
		[0,0]	[1,1]	[0.5,0.7]
		Iteracijų sk.	Iteracijų sk.	Iteracijų sk.
Žingsnio daugikliai γ	0.001	0	1000+	1000+
	0.01	0	1000+	1000+
	0.1	0	475	1000+
	0.003	0	1000+	1000+
	0.03	0	1000+	1000+
	0.3	0	156	359

2 lentelė: Gradientinio nusileidimo metodo algoritmo iteracijų skaičiai visiems γ ir x_0

Pirmas į akis krintantis momentas yra 0 iteracijų pradiniam taškui [0,0]. Galima manyti, kad nuliniam rezultatui apdoroti vis tiek turėtų prireikti bent 1 iteracijos, ir tai nebūtų klaida. Tačiau tai priklauso nuo algoritmo įgyvendinimo. Mano atveju 1 iteraciją laikau baigta, kai apskaičiuojamas kitas bandomasis taškas x_i, vadinasi, galime tęsti toliau. Tačiau šiuo atveju niekada negalėsime tęsti, nes gradiento norma taške [0,0] visada bus lygi 0, t. y. mažesnė už tikslumo reikšmę ε. Kalbant apie kitus taškus, rezultatai yra tikėtini: kaip jau minėjau, dauguma pasirinktų žingsnio daugiklių neduoda gero rezultato, kai neviršijama maksimali iteracijų riba. Net vienas iš didesnių, 0.1, nesugeba neviršyti pradinio taško [0.5,0.7] ribos.

4.1.3 Efektyvumas

Čia yra gradientinio nusileidimo metodo efektyvumo lentelė:

		Nuliniai taškai x_0		
		[0,0]	[1,1]	[0.5,0.7]
		Funkc. iškv. sk.	Funkc. iškv. sk.	Funkc. iškv. sk.
Žingsnio daugikliai γ	0.001	2	1000+	1000+
	0.01	2	1000+	1000+
	0.1	2	477	1000+
	0.003	2	1000+	1000+
	0.03	2	1000+	1000+
	0.3	2	158	361

3 lentelė: Gradientinio nusileidimo metodo algoritmo funkcijų iškvietimų skaičiai visiems γ ir x_0

Metodo efektyvumo lentelė beveik lygiai atitinka greičio lentelę, o skirtumas tarp reikšmių yra 2. Taip yra dėl papildomo gradiento ir funkcijos reikšmės taške skaičiavimo. Pažvelgus į mano kodą nesunku pastebėti, kad gradientas apskaičiuojamas prieš padidinant iteracijų skaitiklį, jis visada yra priekyje vienetu. Funkcijos reikšmės apskaičiavimas paskutiniame taške padidina skirtumą tarp iteracijų ir funkcijos įvertinimų iki dviejų. Vadinasi, todėl kiekviename taške ir kiekviename žingsnio daugikliui γ skirtumas tarp dviejų lentelių bus 2.

Apibendrinant šio metodo rezultatus: populiariausiais laikyti žingsnio daugikliai galiausiai pasirodo esą nelabai veiksmingi mūsų tikslinei funkcijai. Galbūt yra būdas, kaip tokį γ apskaičiuoti mūsų atvejui? Pereikime prie kito metodo - gradientinio nusileidimo.

4.1.4 Tarpiniai rezultatai

		Nuliniai taškai x_0					
		[1,1]			[0.5,0.7]		
		Tarp. taškas	Funkc. reikšmė	Iškv. skaičius	Tarp. taškas	Funkc. reikšmė	Iškv. skaičius
Iteracija	0	[0.975, 0.975]	0.112887	1	[0.493875, 0.694375]	0.00807	1
	50	[0.511251, 0.511251]	0.000735	51	[0.338097, 0.539014]	-0.002799	51
	100	[0.409006, 0.409006]	-0.003806	101	[0.296278, 0.475544]	-0.004019	101
	150	[0.369761, 0.369761]	-0.004452	151	[0.285962, 0.438875]	-0.004317	151
	200	[0.3518, 0.3518]	-0.004585	201	[0.286576, 0.414245]	-0.00444	201
	250	[0.342928, 0.342928]	-0.004618	251	[0.291055, 0.39644]	-0.004507	251
	300	[0.33838, 0.33838]	-0.004626	301	[0.29663, 0.383058]	-0.004549	301
	350	[0.336005, 0.336005]	-0.004629	351	[0.302169, 0.372776]	-0.004577	351
	400	[0.334752, 0.334752]	-0.004629	401	[0.307221, 0.364767]	-0.004595	401
	450	[0.334088, 0.334088]	-0.004629	451	[0.311642, 0.358472]	-0.004607	451
	500	-	-	-	[0.315419, 0.353493]	-0.004614	501
	550	-	-	-	[0.318599, 0.349535]	-0.00462	551
	600	-	-	-	[0.321251, 0.346377]	-0.004623	601
	650	-	-	-	[0.323447, 0.343849]	-0.004625	651
	700	-	-	-	[0.325259, 0.341822]	-0.004627	701
	750	-	-	-	[0.326747, 0.340192]	-0.004628	751
	800	-	-	-	[0.327966, 0.33888]	-0.004628	801
	850	-	-	-	[0.328963, 0.337821]	-0.004629	851
	900	-	-	-	[0.329777, 0.336967]	-0.004629	901
	950	-	-	-	[0.330441, 0.336276]	-0.004629	951
	1000	-	-	-	[0.330982, 0.335718]	-0.004629	1001
	1050	-	-	-	[0.331423, 0.335266]	-0.004629	1051

4 lentelė: Tarpiniai rezultatai kas 50-ąją iteraciją, kai γ=0.1

		Nuliniai taškai x0					
		[1,1]			[0.5,0.7]		
		Tarp. taškas	Funkc. reikšmė	Iškv. skaičius	Tarp. taškas	Funkc. reikšmė	Iškv. skaičius
Iteracija	0	[0.925, 0.925]	0.09091	1	[0.481625, 0.683125]	0.006776	1
	50	[0.366789, 0.366789]	-0.00448	51	[0.28505, 0.436779]	-0.004329	51
	100	[0.337878, 0.337878]	-0.004627	101	[0.296836, 0.38221]	-0.004552	101
	150	[0.333998, 0.333998]	-0.004629	151	[0.311916, 0.358033]	-0.004607	151
	200	-	-	-	[0.321452, 0.346128]	-0.004623	201
	250	-	-	-	[0.326878, 0.340047]	-0.004628	251
	300	-	-	-	[0.329858, 0.336881]	-0.004629	301
	350	-	-	-	[0.331471, 0.335216]	-0.004629	351

5 lentelė: Tarpiniai rezultatai kas 50-ąją iteraciją, kai γ=0.3

4.2 Greičiausio nusileidimo metodo rezultatai ir vizualizacija

Kaip jau minėta, šis metodas veikia identiškai kaip ir ankstesnis, tik atliekamas papildomas žingsnis - auksinio pjūvio algoritmu apskaičiuojamas optimaliausias žingsnio daugiklis γ duotam pradiniam taškui x_0. Dėl to jau dabar galime prognozuoti, kad funkcijos iškvietimų skaičius bus daug didesnis, palyginti su ankstesniuoju metodu. Kartu teoriškai, jei mums pavyks rasti optimalią γ, iteracijų skaičius taip pat turėtų būti minimalus. Pažiūrėkime, ar tai atspindės gauti rezultatai.

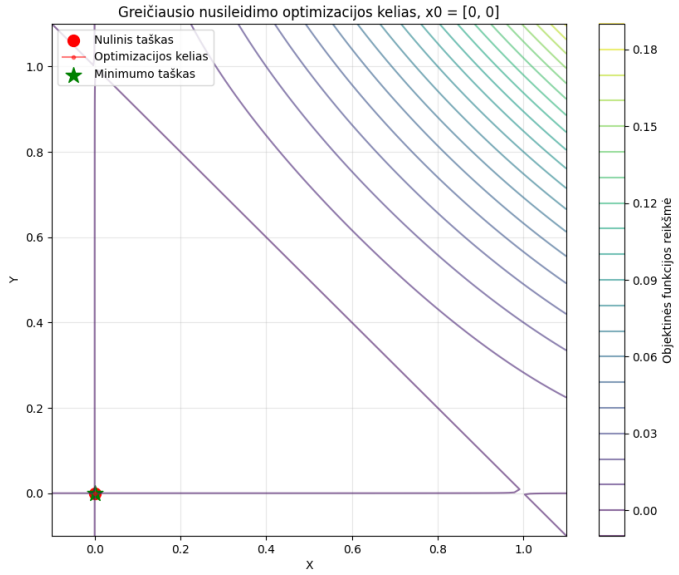
4.2.1 Minimumo taškas ir funkcijos reikšmė

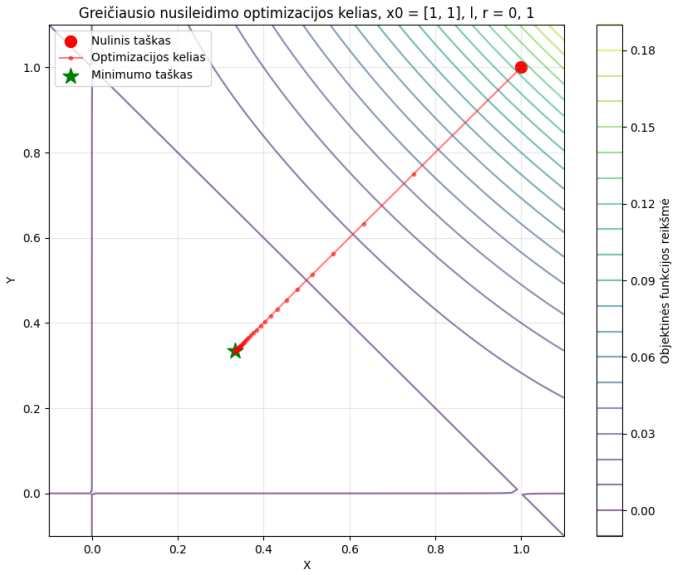
		Auksinio pjūvio algoritmo intervalai					
		[0;1]		[0;7]		[0;20]	
		Min. taškas	Funkc. reikšmė	Min. taškas	Funkc. reikšmė	Min. taškas	Funkc. reikšmė
Nuliniai taškai x_0	[0,0]	[0, 0]	0	[0, 0]	0	[0, 0]	0
	[1,1]	[0.333860, 0.333860]	-0.004629	[0.333330, 0.333330]	-0.004629	-	-
	[0.5,0.7]	[0.331707, 0.334974]	-0.004629	[0.331957, 0.334717]	-0.004629	[0.332230, 0.333935]	-0.004629

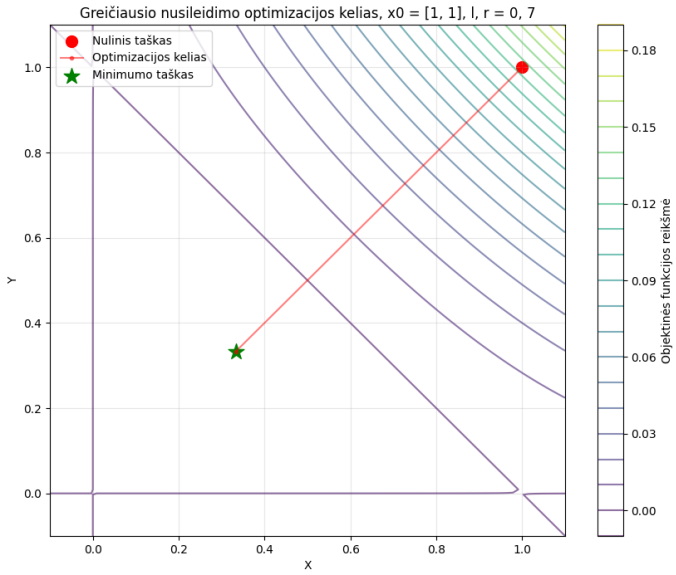
6 lentelė: Greičiausio nusileidimo metodo minimumo taškai ir funkcijos reikšmės visiems auksinio pjūvio algoritmo intervalams ir x_0

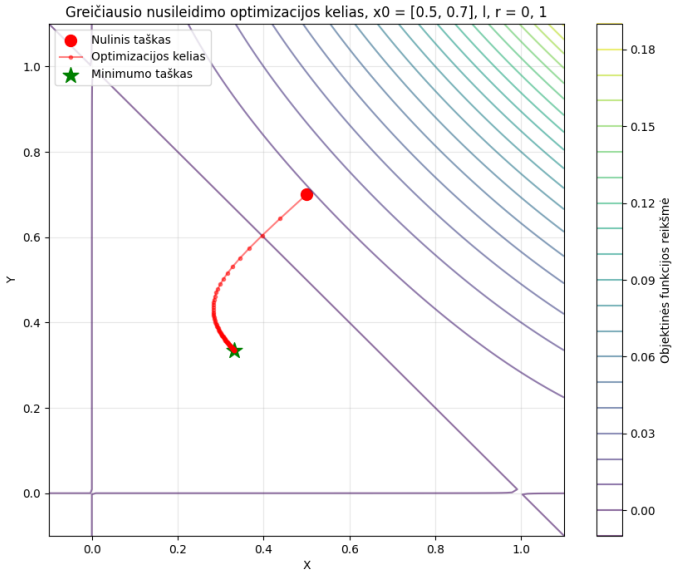
Vėlgi, pradinio taško [0,0] rezultatas yra identiškas ankstesnio metodo rezultatui dėl tos pačios priežasties - to taško gradiento norma yra 0, todėl net nepasieksime nei optimalaus γ skaičiavimo, nei kito bandymo taško. Tačiau rezultatai pradeda atrodyti įdomūs kitiems dviem pradiniams taškams. Šį kartą taip pat reikia atkreipti dėmesį į aukso pjūvio algoritmo intervalą, kitaip tariant, sritį, kurioje ieškome optimalaus žingsnio daugiklio. Pažvelkime į pradinio taško [1,1] rezultatus. Pirmiausia į akis krinta tai, kad nėra [0;20] aukso pjūvio intervalo rezultatų. Optimali γ minėtam pradiniam taškui yra apytiksliai 2,666677. Kaip matyti, mano įgyvendintas šis metodas gerai veikia intervale [0;7]. Tačiau, kai bandau taikyti šį metodą intervalui [0;20], gaunama perpildymo klaida. Taip yra dėl to, kad kai gama tampa didesnė už optimalią, mūsų x_{i+1} bandymo taškai iš (4) funkcijos tampa per dideli. Tačiau [0.5,0.7] taško atveju šis intervalas yra geras dėl to, kad gradientas ten nėra toks didelis kaip ankstesniame taške, todėl galima naudoti didesnį žingsnio daugiklį, kuris šiuo atveju yra 11,319706. Kalbant apie minimumo taškus ir funkcijos reikšmes, jos yra daugiau ar mažiau panašios, su nedideliais nukrypimais.

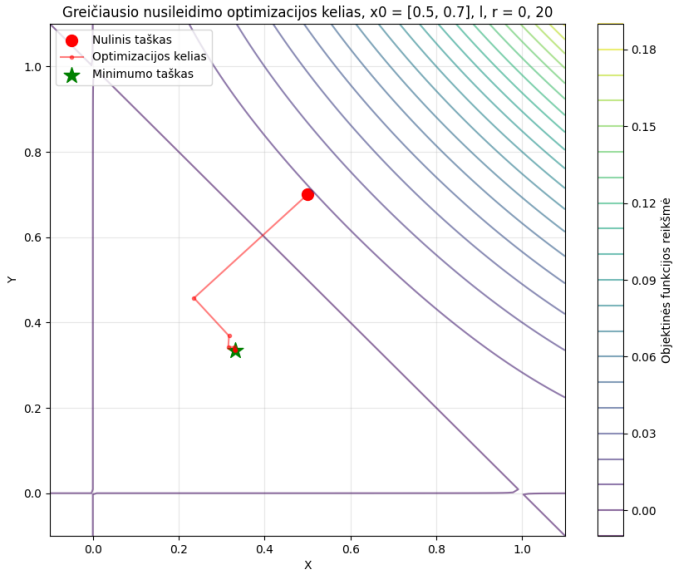
Štai kaip atrodo šio metodo grafikai:


Greičiausio nusileidimo optimizacijos kelias, x0 = [0, 0]
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė


Greičiausio nusileidimo optimizacijos kelias, x0 = [1, 1], l, r = 0, 1
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė


Greičiausio nusileidimo optimizacijos kelias, x0 = [1, 1], l, r = 0, 7
Nulinis taškas
Optimizacijos kelias
Minimumo taškas
X
Y
Objektinės funkcijos reikšmė





Kairėje pusėje esantys grafikai labai panašūs į gradientinio nusileidimo metodo grafikus. Tačiau kaip dėl dešiniųjų grafikų? Kodėl yra tiek mažai bandomųjų taškų? Į šiuos klausimus atsakysiu metodų greičio pastraipoje.

4.2.2 Greitis

		Auksinio pjūvio algoritmo intervalai		
		[0;1]	[0;7]	[0;20]
		Iteracijų sk.	Iteracijų sk.	Iteracijų sk.
Nuliniai taškai x_0	[0,0]	0	0	0
	[1,1]	44	1	-
	[0.5,0.7]	107	14	7

7 lentelė: Greičiausio nusileidimo metodo algoritmo iteracijų skaičiai visiems auksinio pjūvio intervalams ir x_0

Vėlgi, pradinio taško [0,0] rezultatai yra savaime suprantami. Kalbant apie taškus [1,1] ir [0.5,0.7], iteracijų skaičius, palyginti su ankstesniu metodu, jau yra gana mažas. Taip yra todėl, kad galimo optimalaus žingsnio daliklio bazinė sritis [0,1] jau leidžia aukso pjūviui parinkti daug didesnę γ, todėl sumažėja bendras iteracijų skaičius. Tačiau žvelgiant į kitus du intervalus [0;7] ir [0;20], kaip

iteracijų skaičius gali būti toks mažas? Esmė tame, kad taikant greičiausią nusileidimą, kiekvienoje iteracijoje apskaičiuojame gradientą dabartiniame taške ir naudojame jį paieškos krypčiai nustatyti. Tada funkcija minimizuojama tik pagal šią liniją (neigiamo gradiento kryptį). Ši paieška pagal liniją supaprastina optimizavimo procesą, nes, užuot optimizavę sudėtingą daugiamatį paviršių, optimalaus žingsnio daliklio ieškome tik pagal vieną liniją. Štai kodėl gauname tokį mažą iteracijų skaičių intervalams, artimiems optimaliam γ.

4.2.3 Efektyvumas

		Auksinio pjūvio algoritmo intervalai		
		[0;1]	[0;7]	[0;20]
		Funkc. iškv. sk.	Funkc. iškv. sk.	Funkc. iškv. sk.
Nuliniai taškai x_0	[0,0]	2	2	2
	[1,1]	1014	29	-
	[0.5,0.7]	2463	380	205

8 lentelė: Greičiausio nusileidimo metodo algoritmo funkcijų iškvietimų skaičiai visiems auksinio pjūvio intervalams ir x_0

Efektyvumas - pagrindinis greičiausio nusileidimo metodo trūkumas. Kadangi algoritmas atlieka papildomą vienmatį optimizavimą, kad surastų optimalią γ, atitinkamai išauga funkcijos iškvietimų skaičius. Iš esmės tai yra greičio ir efektyvumo kompromisas. Siekiant greitesnių rezultatų, t. y. per kuo mažiau iteracijų, metodas žymiai dažniau kviečia tikslo funkciją ir jos gradientą, todėl tampa mažiau efektyvus.

4.2.4 Tarpiniai rezultatai

		Nuliniai taškai x_0					
		[1,1]			[0.5,0.7]		
Iteracija		Tarp. taškas	Funkc. reikšmė	Iškv. skaičius	Tarp. taškas	Funkc. reikšmė	Iškv. skaičius
	0	[0.750008, 0.750008]	0.035158	23	[0.438752, 0.643752]	0.002913	23
	10	[0.384066, 0.384066]	-0.004275	253	[0.286888, 0.461625]	-0.004163	253
	20	[0.345145, 0.345145]	-0.004612	483	[0.285734, 0.407747]	-0.004464	483
	30	[0.33635, 0.33635]	-0.004628	713	[0.297701, 0.379314]	-0.004559	713
	40	[0.334121, 0.334121]	-0.004629	943	[0.308552, 0.362376]	-0.004599	943
	50	-	-	-	[0.31658, 0.351885]	-0.004617	1173
	60	-	-	-	[0.322164, 0.345269]	-0.004624	1403
	70	-	-	-	[0.325943, 0.341051]	-0.004627	1633
	80	-	-	-	[0.328466, 0.33834]	-0.004629	1863
	90	-	-	-	[0.330137, 0.336589]	-0.004629	2093
	100	-	-	-	[0.331238, 0.335454]	-0.004629	2323

9 lentelė: Tarpiniai rezultatai kas 10-ąją iteraciją, kai auksinio pjūvio algoritmo intervalas yra [0;1]

4.3 Deformuojamo simplekso (Nelder-Mead) metodo rezultatai ir vizualizacija

Skirtingai nei ankstesni du metodai, deformuojamas simpleksas veikia tik su figūromis ir visiškai nesiremia tikslo funkcijos gradientu. Tačiau ar jis gali prilygti jų tikslumui? Pažvelkime į rezultatus.

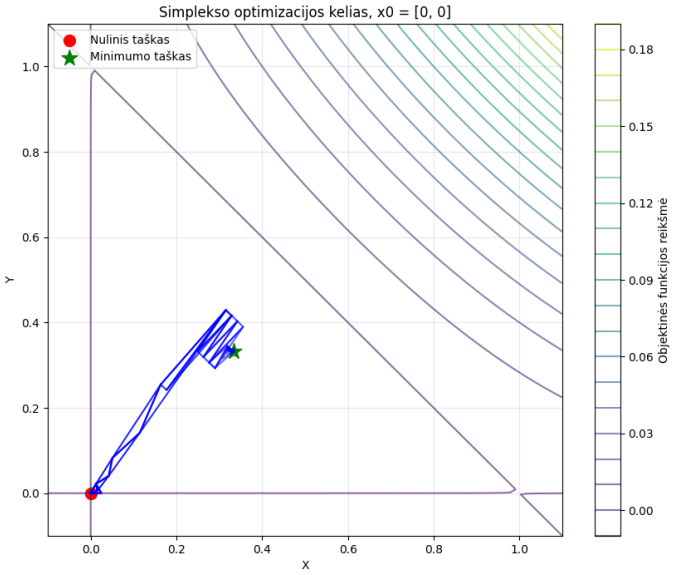
4.3.1 Minimumo taškas ir funkcijos reikšmė

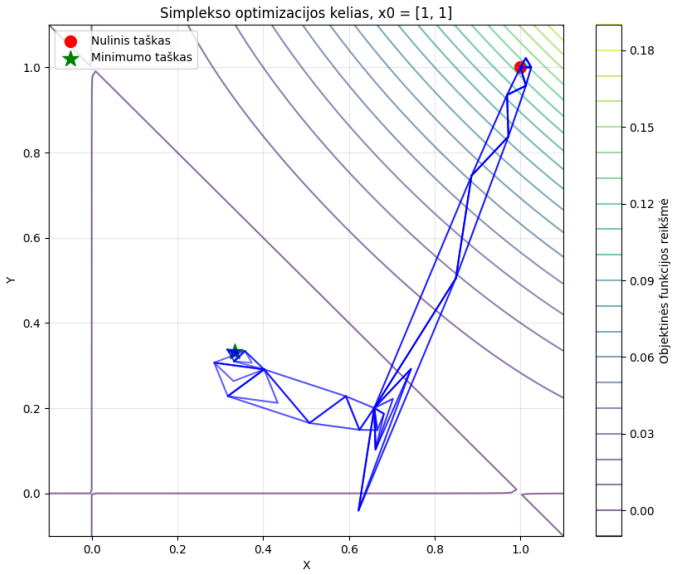
Nuliniai taškai x_0	Min. taškas	Funkc. reikšmė
[0,0]	[0.333282, 0.333380]	-0.004629
[1,1]	[0.333349, 0.333354]	-0.004629
[0.5,0.7]	[0.333368, 0.333323]	-0.004629

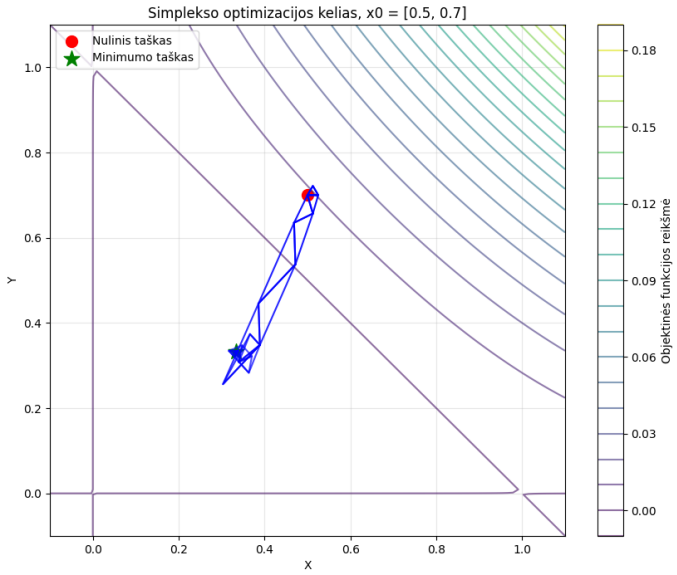
10 lentelė: Deformuojamo simplekso metodo algoritmo minimumo taškai ir funkcijos reikšmės visiems x_0

Minimumo taškai ir funkcijos reikšmės yra gana panašūs, tik su labai nedideliais nuokrypiais. Apskritai nieko netikėto.

Štaip kaip atrodo deformuojamo simplekso metodo grafikai:







Matome, kad kiekvieną kartą algoritmas pradeda nuo lygiakraščio trikampio, daro atspindį nuo blogiausio taško, plečiasi ir traukiasi, kaip numatyta, kol priartėja prie minimumo taško.

4.3.2 Greitis

Nuliniai taškai x_0	Iteracijų sk.
[0,0]	38
[1,1]	51
[0.5,0.7]	35

11 lentelė: Deformuojamo simplekso metodo algoritmo iteracijų skaičiai visiems x_0

Kaip matome, iteracijų skaičius taip pat yra gana tikėtinas, o tolimiausiam pradiniam taškui [1,1] pasiekti minimumo prireikė daugiausiai iteracijų - 51.

4.3.3 Efektyvumas

Nuliniai taškai x_0	Funkc. iškv. sk.
[0,0]	76
[1,1]	96
[0.5,0.7]	70

12 lentelė: Deformuojamo simplekso metodo algoritmo funkcijų iškvietimų skaičiai visiems x_0

Įdomu, kad kalbant apie efektyvumą, pradiniame taške [0,0] atliekama šiek tiek daugiau funkcijų iškvietimų nei [0.5,0.7], o kalbant apie greitį - atvirkščiai - [0.5,0.7] reikia mažiau iteracijų.

4.3.4 Tarpiniai rezultatai

		Nuliniai taškai x0								
		[0,0]			[1,1]			[0.5,0.7]		
Iteracija		Tarp. taškas	Funkc. reikšmė	Iškv. skaičius	Tarp. taškas	Funkc. reikšmė	Iškv. skaičius	Tarp. taškas	Funkc. reikšmė	Iškv. skaičius
	0	[0.0125, 0.02165]	-0.000033	5	[1.0, 1.0]	0.018966	5	[0.5, 0.7]	0.00875	5
	5	[0.163477, 0.254726]	-0.003028	15	[0.849219, 0.504756]	-0.002322	15	[0.389062, 0.348188]	-0.004449	14
	10	[0.262946, 0.319052]	-0.004383	25	[0.658984, 0.200303]	-0.003426	24	[0.340332, 0.308609]	-0.004609	22
	15	[0.319406, 0.344682]	-0.004623	34	[0.508032, 0.165037]	-0.004488	34	[0.33811, 0.334572]	-0.004628	32
	20	[0.332858, 0.331616]	-0.004629	43	[0.402258, 0.29092]	-0.004628	42	[0.335327, 0.332545]	-0.004629	42
	25	[0.33486, 0.332305]	-0.004629	53	[0.340947, 0.328868]	-0.004629	52	[0.333821, 0.333206]	-0.004629	52
	30	[0.333756, 0.333022]	-0.004629	63	[0.33239, 0.332543]	-0.004629	61	[0.333415, 0.333198]	-0.004629	62
	35	[0.333283, 0.33338]	-0.004629	73	[0.334782, 0.332384]	-0.004629	71	-	-	-
	40	-	-	-	[0.333627, 0.333317]	-0.004629	78	-	-	-
	45	-	-	-	[0.333373, 0.333386]	-0.004629	87	-	-	-

13 lentelė: Tarpiniai rezultatai kas 5-ąją iteraciją

4.4 Trijų algoritmų palyginamoji analizė

Visi trys optimizavimo be apribojimų metodai turi savų privalumų ir trūkumų. Nors gradiento nusileidimo metodą palyginti lengva įgyvendinti kodu, jį taikant reikia rankiniu būdu įvesti žingsnio daugiklį, o tai, kaip matėme, ne visada labai efektyvu. Kartais galima iš karto atspėti gerą reikšmę, gaunant solidų iteracijų ir funkcijų iškvietimų kiekį, neviršijantį 100, kitais atvejais atspėta γ gali priversti algoritmą „peržengti“ minimumo tašką ir sukelti klaidą. Tai iš esmės verčia žmones žaisti spėjimo žaidimą su γ kintamuoju, o tai gana nepatogu ir atima daug laiko.

Greičiausio nusileidimo metodas yra ankstesnio patobulinimas. Užuot spėliojus γ reikšmę, algoritmas atlieka papildomą vienmatį optimizavimą, kad rastų optimalią γ reikšmę, o tai dažniausiai duoda neįtikėtinai mažą iteracijų skaičių. Tačiau tas papildomas optimizavimas galiausiai gerokai padidina bendrą funkcijų iškvietimų skaičių, todėl šis metodas yra gana neefektyvus. Be to, jį šiek tiek sudėtingiau įgyvendinti ir vėliau analizuoti, priklausomai nuo vidinio vienmačio optimizavimo algoritmo, kuris mano atveju buvo auksinis pjūvis. Dėl to atsirado dar vienas kintamasis, į kurį turėjau atsižvelgti rašydamas ataskaitą, - auksinio pjūvio algoritmo intervalas. Tuo metu šį algoritmą pasirinkau todėl, kad dar turėjau šalia jo realizaciją, tačiau paaiškėjo, kad dirbti su juo gana sudėtinga, nes nors techniškai nereikėtų pateikti jokio intervalo kaip argumento, kaip jau matėme, rezultatai vis tiek nuo jo priklauso. Taigi bet kuris tyrėjas, bandantis įgyvendinti šį optimizavimo be apribojimų metodą, prieš rašydamas paties metodo realizaciją turėtų būtinai apgalvoti vidinį vienmatį optimizavimo algoritmą.

Deformuojamo simplekso metodas, mano nuomone, yra geras tarpinis variantas tarp dviejų ankstesnių. Jis nepriklauso nuo gradiento, todėl tampa daug nuoseklesnis nepriklausomai nuo pačios tikslo funkcijos. Palyginti su ankstesniais dviem metodais, jis nereikalauja jokių spėlionių ir duoda palyginti greitus ir išteklius taupančius rezultatus. Vienintelis trūkumas, kurį radau, yra sudėtingumas bandant parašyti savo įgyvendinimą. Turėjau net priversti jį pradėti nuo lygiakraščio trikampio, kai jis veikia antrajame matmenyje, kitaip tariant, nuo taško su dviem kintamaisiais, o to programuotojams reikia vengti - bandyti kietai užkoduoti per daug dalykų.

5 Išvada

Apibendrinant, pagrindiniai šio laboratorinio darbo tikslai:

- išanalizuoti šiuos tris optimizavimo be apribojimų metodus - gradientinio nusileidimo, stačiausio nusileidimo ir deformuoto simplekso: jų išvestus minimalius taškus, funkcijų reikšmes, iteracijų kiekį (greitį) ir funkcijų iškvietimų kiekį (efektyvumą);
- išspręsti optimizavimo uždavinį: **Kokia turėtų būti stačiakampio gretasienio formos dėžė, kad vienetiniam paviršiaus plotui jos tūris būtų maksimalus?**

Po išsamios analizės galima pateikti atsakymą: **dėžė turi būti kubas, kurio viena į kitą atgręžtų sienų plotų sumos, kurių yra 3, lygūs $\frac{1}{3}$ paviršiaus vieneto ploto, t.y $2ab = 2bc = 2ac = \frac{1}{3}$, kur a, b, c - atitinkamai ilgis, plotis ir aukštis**.

6 Priedas

imports.py failas:

```
import math
from datetime import datetime
import sympy as sp
import numpy as np
import scipy as scp
import matplotlib.pyplot as plt
```

objfunc.py failas:

```
class ObjectiveFunction:
    counter = 0
```

```
    def __init__(self):
        pass

    def f(self, x):
        ObjectiveFunction.counter += 1
        return -1/8*(x[0]*x[1]-x[0]**2*x[1]-x[0]*x[1]**2)

    def gradF(self, x):
        ObjectiveFunction.counter += 1
        return [-1/8*(x[1]-2*x[0]*x[1]-x[1]**2), -1/8*(x[0]-2*x[0]*x[1]-x[0]**2)]

    def reset(self):
        ObjectiveFunction.counter = 0
```

plotfunc.py failas (siek tiek turėjau pakeisti kad galėčiau įkelti):

```
from imports import plt, np
from objfunc import ObjectiveFunction

def graph(obj:ObjectiveFunction, initialPoint, points, methodId, gamma=0, interv=[0,1
                                                ]):
    match methodId:
        case 1:
            methodName = f'Gradientinio nusileidimo optimizacijos kelias, x0 = {
                                                    initialPoint}, gamma = {gamma}
                                                    '
        case 2:
            methodName = f'Greičiausio nusileidimo optimizacijos kelias, x0 = {
                                                    initialPoint}, l, r = {interv[
                                                    0]}, {interv[1]}'
        case 3:
            methodName = f'Simplekso optimizacijos kelias, x0 = {initialPoint}'

    x = np.linspace(-0.1, 1.1, 100)
    y = np.linspace(-0.1, 1.1, 100)
    X, Y = np.meshgrid(x, y)

    Z = np.zeros_like(X)
    for i in range(len(x)):
        for j in range(len(y)):
            Z[i,j] = obj.f([X[i,j], Y[i,j]])

    plt.figure(figsize=(10, 8))
    plt.contour(X, Y, Z, levels=20, cmap='viridis', alpha=0.6)

    if methodId==1 or methodId==2:
        points = np.array(points)
        plt.scatter(initialPoint[0], initialPoint[1], color='red', marker='o', s=100,
                                                label='Nulinis taškas')
        plt.plot(points[:,0], points[:,1], 'r.-', alpha=0.5, label='Optimizacijos
                                                kelias')
        plt.scatter(points[-1,0], points[-1,1], color='green', marker='*', s=200,
                                                label='Minimumo taškas')

    elif methodId == 3:
        plt.scatter(initialPoint[0], initialPoint[1], color='red', marker='o', s=100,
                                                label='Nulinis taškas')

        for i, triangle in enumerate(points):
```

```
            alpha = max(0.1, 1 - i/len(points))
            triangle = np.array(triangle)
            plt.plot([triangle[0,0], triangle[1,0], triangle[2,0], triangle[0,0]],
                     [triangle[0,1], triangle[1,1], triangle[2,1], triangle[0,1]],
                     'b-', alpha=alpha)

        final_triangle = np.array(points[-1])
        plt.scatter(final_triangle[0,0], final_triangle[0,1], color='green', marker='
                                                *', s=200, label='Minimumo taškas'
                                                )

    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title(f'{methodName}')
    plt.legend(loc=2)
    plt.grid(True, alpha=0.3)
    plt.show()
```

gradDescent.py failas:

```
from imports import np
from objfunc import ObjectiveFunction

def f(x):
    return -1/8*(x[0]*x[1]-x[0]**2*x[1]-x[0]*x[1]**2)

def gradDescent(obj:ObjectiveFunction, x0, gamma, eps=1e-4, maxIter=10000):
    iter = 0
    x = x0.copy()
    points = [x0.copy()]

    while iter < maxIter:
        grad = obj.gradF(x)
        if np.linalg.norm(grad) < eps:
            break

        x = [x[j] - gamma * grad[j] for j in range(len(x))]
        points.append(x.copy())

        '''if iter%50==0:
            print(f"[{iter}]: x = {x}, f(x) = {f(x)}, func. calls = {obj.counter}")''
                                                    '


        iter += 1

    fX = obj.f(x)

    return x, fX, iter, points
```

steepDescent.py failas:

```
from objfunc import ObjectiveFunction
from imports import np
import math

def f(x):
    return -1/8*(x[0]*x[1]-x[0]**2*x[1]-x[0]*x[1]**2)

def steepDescent(obj:ObjectiveFunction, x0, eps=1e-4, maxIter=200):
```

```
    iter = 0
    x = x0.copy()
    points = [x0.copy()]
    optimalGamma = 0
    while iter < maxIter:
        grad = obj.gradF(x)

        if np.linalg.norm(grad) < eps:
            break

        def objective(gamma):
            return obj.f([x[j] - gamma * grad[j] for j in range(len(x))])

        l, r = 0, 1    # Vėliau taip pat bus nagrinėjami l,r = 0,7 bei l,r = 0,20
        tau = (math.sqrt(5)-1)/2
        L = r-l
        x1 = l+(1-tau)*L
        x2 = l+tau*L
        fx1 = objective(x1)
        fx2 = objective(x2)

        while (r-l) > eps:
            if fx1 < fx2:
                r = x2
                x2 = x1
                fx2 = fx1
                L = r-l
                x1 = l+(1-tau)*L
                fx1 = objective(x1)
            else:
                l = x1
                x1 = x2
                fx1 = fx2
                L = r-l
                x2 = l+tau*L
                fx2 = objective(x2)

        gammaOpt = (l+r)/2
        optimalGamma = gammaOpt

        x = [x[j]-gammaOpt*grad[j] for j in range(len(x))]
        points.append(x.copy())

        '''if iter%10==0:
            print(f"[{iter}]: x = {x}, f(x) = {f(x)}, func. calls = {obj.counter}")''
                                                                    '


        iter += 1

    print(optimalGamma)
    return x, obj.f(x), iter, points
```

simplex.py failas:

```
from objfunc import ObjectiveFunction

def f(x):
    return -1/8*(x[0]*x[1]-x[0]**2*x[1]-x[0]*x[1]**2)

def simplex(obj:ObjectiveFunction, x0, initStep=0.025, stepCoeff=1.025, eps=1e-4,
```

```
                                        alpha=1, gamma=2, rho=0.5, maxIter=200):
    n = len(x0)
    pond = [x0]
    fVals = [obj.f(x0)]
    it = 0

    triangles = []

    if n == 2:
        pond.append([x0[0] + initStep, x0[1]])
        fVals.append(obj.f(pond[-1]))

        pond.append([x0[0] + initStep/2, x0[1] + initStep * 0.866])
        fVals.append(obj.f(pond[-1]))
    else:
        for i in range(n):
            xTemp = x0.copy()
            if xTemp[i] == 0:
                xTemp[i] = initStep
            else:
                xTemp[i] *= stepCoeff
            pond.append(xTemp)
            fVals.append(obj.f(xTemp))

    triangles.append(pond.copy())

    while True and it < maxIter:
        it += 1

        sortedInd = sorted(range(len(fVals)), key=lambda i: fVals[i])
        pond = [pond[i] for i in sortedInd]
        fVals = [fVals[i] for i in sortedInd]

        triangles.append(pond.copy())

        x_m = [sum([pond[i][j] for i in range(len(pond) - 1)]) / n for j in range(n)]

        x_r = [x_m[j] + alpha * (x_m[j] - pond[-1][j]) for j in range(n)]
        y_r = obj.f(x_r)

        if fVals[0] <= y_r < fVals[-2]:
            pond[-1], fVals[-1] = x_r, y_r
        elif y_r < fVals[0]:
            x_e = [x_m[j] + gamma * (x_m[j] - pond[-1][j]) for j in range(n)]
            y_e = obj.f(x_e)
            if y_e < y_r:
                pond[-1], fVals[-1] = x_e, y_e
            else:
                pond[-1], fVals[-1] = x_r, y_r
        else:
            x_c = [x_m[j] + rho * (pond[-1][j] - x_m[j]) for j in range(n)]
            y_c = obj.f(x_c)
            if y_c < fVals[-1]:
                pond[-1], fVals[-1] = x_c, y_c

        xErr = max([sum((pond[i][j] - pond[0][j]) ** 2 for j in range(n)) ** 0.5 for
                                                    i in range(1, len(pond))])
        yErr = abs(fVals[0] - fVals[-1])

        if xErr < eps and yErr < eps:
```

```
            break

        #print(it)
        if (it-1)%5==0:
            print(f"[{it-1}]: x = {pond[0]}, f(x) = {f(pond[0])}, func. calls = {obj.
                                                    counter}")

    return pond[0], fVals[0], it, triangles
```

main.py failas:

```
from imports import datetime, math, np, sp, plt, scp
from objfunc import ObjectiveFunction
from plotfunc import graph
from gradDescent import gradDescent
from steepDescent import steepDescent
from simplex import simplex

print(f"{datetime.now()}\n")

obj = ObjectiveFunction()
point = [0.5,0.7]
gamma = 0.3 # Gammas to analyze:  0.001, 0.003, 0.01, 0.03, 0.1, 0.3

a, b, c, points = gradDescent(obj, point, gamma)
print(f"Minimum point: {a}")
print(f"Value @ min. point: {b}")
print(f"nit: {c}")
print(f"nfev: {obj.counter}")

#graph(obj, point, points, 1, gamma)

obj.reset()
print("----------------------[STEEP DESCENT]-------------------------")
a, b, c, points = steepDescent(obj, point)
print(f"Minimum point: {a}")
print(f"Value @ min. point: {b}")
print(f"nit: {c}")
print(f"nfev: {obj.counter}")

#graph(obj, point, points, 2, 0, [0,20])

obj.reset()
print("----------------------[SIMPLEX SciPy]-------------------------")
print(scp.optimize.minimize(obj.f, point, method='Nelder-Mead'))

obj.reset()
print("----------------------[SIMPLEX CUSTOM]-------------------------")
a, b, c, points = simplex(obj, point)
print(f"Minimum point: {a}")
print(f"Value @ min. point: {b}")
print(f"nit: {c}")
print(f"nfev: {obj.counter}")

#graph(obj, point, points, 3)

obj.reset()
```