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 gradDescent(obj:ObjectiveFunction, x0, gamma, eps=1e-4, maxIter=1000):
    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())
        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. 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 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())
        iter += 1

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

3.3 Deformuojamo simplekso metodas (Nelder-Mead)

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 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

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

4 Rezultatai ir jų analyzė

4.1 Gradientinio nusileidimo metodo rezultatai ir vizualizacija

4.2 Greičiausio nusileidimo metodo rezultatai ir vizualizacija

4.3 Deformuojamo simplekso metodo rezultatai ir vizualizacija

4.4 Trijų algoritmų palyginamoji analizė

5 Išvada

6 Priedas