KAUNO TECHNOLOGIJOS UNIVERSITETAS
Informatikos fakultetas

P170B115 Skaitiniai metodai ir algoritmai

2 LABORATORINIS DARBAS

2023-11-30

Atliko: IFF-1/6 gr. studentas Lukas Kuzmickas

Priėmė: Doc. Andrius Kriščiūnas

KAUNAS, 2023

Turinys

1. Tiesinių lygčių sistemų sprendimas

a) Lentelėje 1 duotos tiesinės lygčių sistemos, 2 lentelėje nurodyti metodai ir lygčių sistemų numeriai (iš 1 lentelės). Reikia suprogramuoti nurodytus metodus ir jais išspręsti pateiktas lygčių sistemas.

Sprendžiant lygčių sistemas (a ir b punktuose), turi būti:

a) Programoje turi būti įvertinti atvejai:
   • kai lygčių sistema turi vieną sprendinį;
   • kai lygčių sistema sprendinių neturi;
   • kai lygčių sistema turi be gali daug sprendinių.
b) Patikrinkite gautus sprendinius ir skaidas, įrašydami juos į pradinę lygčių sistemą.
c) Gautą sprendinį patikrinkite naudodami išorinius išteklius (pvz., standartines Python funkcijas).

10	Atspindžio	8, 13, 20
	Paprastųjų iteracijų	8

1 pav. Užduoties variantas.

1.1. 8 lygčių sistema. Atspindžio metodas (2 pav.)

8	$\begin{cases} 4x_1 + 3x_2 - x_3 + x_4 = 12 \\ 3x_1 + 9x_2 - 2x_3 - 2x_4 = 10 \\ -x_1 - 2x_2 + 11x_3 - x_4 = -28 \\ x_1 - 2x_2 - x_3 + 5x_4 = 16 \end{cases}$

2 pav. 8 lygties sistema.

Iš turimos lygčių sistemos padarome koeficientų matricą A ir laisvųjų narių matricą B.

Matrica A
4,3,-1,1
3,9,-2,-2
-1,-2,11,-1
1,-2,-1,5

Laisvųjų narių matrica B
12
10
-28
16

Kiekvienas stulpelis yra laikomas vektoriumi taikant Atspindžio algoritmą. Surandame mūsų z paprastą ir atspindėtą vektorius.

Paprastas z vektorius = 4 3 -1 1
Atspindėtas z vektorius z' = $\sqrt{4^2 + 3^2 + (-1)^2 + 1^2} =$

$$\begin{matrix} 5.19 \\ 0 \\ 0 \\ 0 \end{matrix}$$

Algoritme pertvarkomi ir kiti matricos stulpeliai panašiu principu.

Apskaičiuojame omega (normalės vektorių) pagal lygtį $w = \frac{z - z'}{[\![z - z']\!]}$

Žinome, kad kiekviena atspindžio matrica yra simetrinė ir ortogonalioji.

$$Q = E - 2ww^T, apskaičiuojame\ atspindžio\ matricą$$

```
A1 matrica:
[[  5.2    7.51  -4.23   0.77  23.48]
 [ -0.     6.45  -1.43  -4.46  -4.06]
 [  0.     0.    10.35  -1.56 -25.36]
 [  0.     0.    -0.    -2.85  -8.54]]
```

3 pav. Atspindžio matrica.

Kai turime matricą Q, atliekame atgalinį Gauso algoritmo etapą:

```
X sprendimų matrica:
[[ 1.]
 [ 1.]
 [-2.]
 [ 3.]]
```

4 pav. Gauso algoritmo atgalinio etapo rezultatas.

Patikrinimui naudojame Python funkciją np.linalg.solve(), kuri apskaičiuoja mums jau šitą matricą.

```
Atsakymas su išoriniais ištekliais:
[[ 1.]
 [ 1.]
 [-2.]
 [ 3.]]
```

5 pav. np.linalg.solve() rezultatai.

Įsistatome šiuos rezultatus į mūsų lygtis.

$$4x_1 + 3x_2 - x_3 + x_4 = 12$$
$$3x_1 + 9x_2 - 2x_3 - 2x_4 = 10$$
$$-x_1 - 2x_2 + 11x_3 - x_4 = -28$$
$$x_1 - 2x_2 - x_3 + 5x_4 = 16$$

```
Laisvųjų narių matrica sudauginus A ir x matricas
[[ 12.]
 [ 10.]
 [-28.]
 [ 16.]]
Pradinė B laisvųjų narių matrica
[[ 12.]
 [ 10.]
 [-28.]
 [ 16.]]
```

Atsakymai atitinka mūsų koeficientų matricos rezultatus.

1.2. 13 lygčių sistema. Atspindžio metodas (3 pav.)

13	$\begin{cases} x_1 - 2x_2 + 3x_3 + 4x_4 = 11 \\ x_1 - x_3 + x_4 = -4 \\ 2x_1 - 2x_2 + 2x_3 + 5x_4 = 7 \\ -7x_2 + 3x_3 + x_4 = 2 \end{cases}$

6 pav. 13 lygties sistema.

Iš turimos lygčių sistemos padarome koeficientų matricą A ir laisvųjų narių matricą B.

Matrica A
1,-2,3,4
1,0,-1,1
2,-2,2,5
0,-7,3,5

Laisvųjų narių matrica B
11
-4
7
2

Kiekvienas stulpelis yra laikomas vektoriumi taikant Atspindžio algoritmą. Surandame mūsų z paprastą ir atspindėtą vektorius.

Paprastas z vektorius = 1 1 2 0
Atspindėtas z vektorius z' = $\sqrt{1^2 + 1^2 + (2)^2 + 0^2} =$

$$\begin{matrix} 2.44 \\ 0 \\ 0 \\ 0 \end{matrix}$$

Algoritme pertvarkomi ir kiti matricos stulpeliai panašiu principu.

Apskaičiuojame omega (normalės vektorių) pagal lygtį $w = \frac{z - z'}{[\![z - z']\!]}$

Žinome, kad kiekviena atspindžio matrica yra simetrinė ir ortogonalioji.

$$Q = E - 2ww^T, apskaičiuojame\ atspindžio\ matricą$$

```
A1 matrica:
[[ 2.45 -2.45  1.63  6.12  8.57]
 [ 0.    7.14 -3.78 -1.4  -4.06]
 [-0.    0.    3.75  1.79  9.51]
 [-0.   -0.    0.    0.58  3.08]]
```

7 pav. Atspindžio matrica.

Kai turime matricą Q, atliekame atgalinį Gauso algoritmo etapą:

```
X sprendimų matrica:
[[-9.32]
 [ 0.47]
 [ 0.  ]
 [ 5.32]]
```

8 pav. Gauso algoritmo atgalinio etapo rezultatas.

Patikrinimui naudojame Python funkciją np.linalg.solve(), kuri apskaičiuoja mums jau šitą matricą.

```
Atsakymas su išoriniais ištekliais:
[[-9.32]
 [ 0.47]
 [-0.  ]
 [ 5.32]]
```

9 pav. np.linalg.solve() rezultatai.

Įsistatome šiuos rezultatus į mūsų lygtis.

```
Laisvųjų narių matrica sudauginus A ir x matricas
[[11.]
 [-4.]
 [ 7.]
 [ 2.]]
Pradinė B laisvųjų narių matrica
[[11.]
 [-4.]
 [ 7.]
 [ 2.]]
```

Atsakymai yra arti teisingo rezultato.

1.3. 20 lygčių sistema. Atspindžio metodas (4 pav.)

20	$\begin{cases} 2x_1 + 4x_2 + 6x_3 - 2x_4 = 2 \\ x_1 + 3x_2 + x_3 - 3x_4 = 1 \\ x_1 + x_2 + 5x_3 + x_4 = 7 \\ 2x_1 + 3x_2 - 3x_3 - 2x_4 = 2 \end{cases}$

10 pav. 20 lygties sistema.

Iš turimos lygčių sistemos padarome koeficientų matricą A ir laisvųjų narių matricą B.

Matrica A
2,4,6,-2
1,3,1,-3
1,2,5,1
2,3,-3,-2

Laisvųjų narių matrica B
2
1
7
2

Kiekvienas stulpelis yra laikomas vektoriumi taikant Atspindžio algoritmą. Surandame mūsų z paprastą ir atspindėtą vektorius.

Paprastas z vektorius = 2 1 1 2
Atspindėtas z vektorius z' = $\sqrt{2^2 + 1^2 + (1)^2 + 2^2} =$

$$\begin{matrix} 3.16 \\ 0 \\ 0 \\ 0 \end{matrix}$$

Algoritme pertvarkomi ir kiti matricos stulpeliai panašiu principu.

Apskaičiuojame normalės vektorių pagal lygtį $w = \frac{z - z'}{[\![z - z']\!]}$

Žinome, kad kiekviena atspindžio matrica yra simetrinė ir ortogonalioji.

$$Q = E - 2ww^T, apskaičiuojame\ atspindžio\ matricą$$

```
A1 matrica:
[[ 3.16  5.69  3.79 -3.16  5.06]
 [ 0.    1.61  0.87 -2.48 -2.98]
 [ 0.    0.    7.47  1.36  3.4 ]
 [ 0.   -0.   -0.   -0.   -3.46]]
```

11 pav. Atspindžio matrica.

Kai turime matricą Q, atliekame atgalinį Gauso algoritmo etapą:

```
X sprendimų matrica:
[[-1.31e+18]
 [ 1.24e+18]
 [-1.37e+17]
 [ 7.56e+17]]
```

12 pav. Gauso algoritmo atgalinio etapo rezultatas.

Patikrinimui naudojame Python funkciją np.linalg.solve(), kuri apskaičiuoja mums jau šitą matricą.

```
LinAlgError: Singular matrix
```

13 pav. np.linalg.solve() rezultatai.

Mūsų Python funkcija, pasako, kad mūsų apskaičiuojama matrica yra singuliari, todėl remiames liekana (rezultatas tarp lygčių sistemos dešinės pusės b koeficientų ir rezultato). Jeigu šios matricos elementai yra labai maži, tai galime daryti išvadą, kad mūsų gauti x matricos rezultatai geri.

```
Liekana:
[[-514.]
 [-257.]
 [-263.]
 [-514.]]
```

14 pav. Liekanos rezultatai.

Mūsų liekanos matricos rezultatai yra gan dideli skaičiai, todėl darome išvadą, kad mūsų matrica yra singuliari, t.y. turi labai daug sprendinių arba jų neturi.

Patikriname santykinę paklaidą. Paklaida labai mažas skaičius, bet vistiek darome išvadą, kad mūsų matrica yra singuliari.

```
Santykine paklaida
4.166639992330018e-16
```

15 pav. Santykinė paklaida.

Įsistatome šiuos rezultatus į mūsų lygtis.

```
Laisvųjų narių matrica sudauginus A ir x matricas
[[-512.]
 [-256.]
 [-256.]
 [-512.]]
Pradinė B laisvųjų narių matrica
[[2.]
 [1.]
 [7.]
 [2.]]
```

1.4. Pirmosios dalies atspindžio metodo kodo fragmentas

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

#1 dalis
np.set_printoptions(precision=2, suppress=True, linewidth=120)

# # #Matrica 8 lygciu sistemos
# A=np.matrix([[4,3,-1,1],
#              [3,9,-2,-2],
#              [-1,-2,11,-1],
#              [1,-2,-1,5]]).astype(float)        # koeficientu matrica
# b=(np.matrix([12,10,-28,16])).transpose().astype(float)   #laisvuju nariu vektorius-
stulpelis

#Matrica 13 lygciu sistemos
# A=np.matrix([[1,-2,3,4],
#              [1,0,-3,1],
#              [2,-2,2,5],
#              [0,-7,3,1]]).astype(float)        # koeficientu matrica
# b=(np.matrix([11,-4,7,2])).transpose().astype(float)   #laisvuju nariu vektorius-
stulpelis

# #Matrica 20 lygciu sistemos
A=np.matrix([[2,4,6,-2],
[1,3,1,-3],
[1,1,5,1],
[2,3,-3,-2]]).astype(float)        # koeficientu matrica
b=(np.matrix([2,1,7,2])).transpose().astype(float)   #laisvuju nariu vektorius-
stulpelis

n=(np.shape(A))[0]   # lygciu skaicius nustatomas pagal ivesta matrica A
nb=(np.shape(b))[1]  # laisvuju nariu vektoriu skaicius nustatomas pagal ivesta
matrica b

A1=np.hstack((A,b))  #isplestoji matrica


# tiesioginis etapas(atspindziai):
for i in range (0,n-1):
z=A1[i:n,i]
zp=np.zeros(np.shape(z)); zp[0]=np.linalg.norm(z)
omega=z-zp; omega=omega/np.linalg.norm(omega)
Q=np.identity(n-i)-2*omega*omega.transpose()
A1[i:n,:]=Q.dot(A1[i:n,:])

# atgalinis etapas (gauso):
x=np.zeros(shape=(n,nb))
for i in range (n-1,-1,-1):    # range pradeda n-1 ir baigia 0 (trecias parametras yra
zingsnis)
    x[i,:]=(A1[i,n:n+nb]-A1[i,i+1:n]*x[i+1:n,:])/A1[i,i]

liekana=A.dot(x)-b;

print("\nA1 matrica:")
print(A1)
print("\nX sprendimų matrica:")
print(x)
print("\nLiekana:")
print(liekana)
print("Santykine paklaida")
print(np.linalg.norm(liekana)/ np.linalg.norm(x))
print("\nAtsakymas su išoriniais ištekliais:")
print(np.linalg.solve(A, b))
```

1.5. 8 lygčių sistema. Paprastųjų iteracijų metodas (5 pav.)

8	$\begin{cases} 4x_1 + 3x_2 - x_3 + x_4 = 12 \\ 3x_1 + 9x_2 - 2x_3 - 2x_4 = 10 \\ -x_1 - 2x_2 + 11x_3 - x_4 = -28 \\ x_1 - 2x_2 - x_3 + 5x_4 = 16 \end{cases}$

16 pav. 8 lygties sistema.

Pradinė matrica A	Pradinė matrica B (laisvųjų narių matrica)
4, 3, -1, 1	12
3, 9, -2, -2	10
-1, -2, 11, -1	-28
1, -2, -1, 5	16

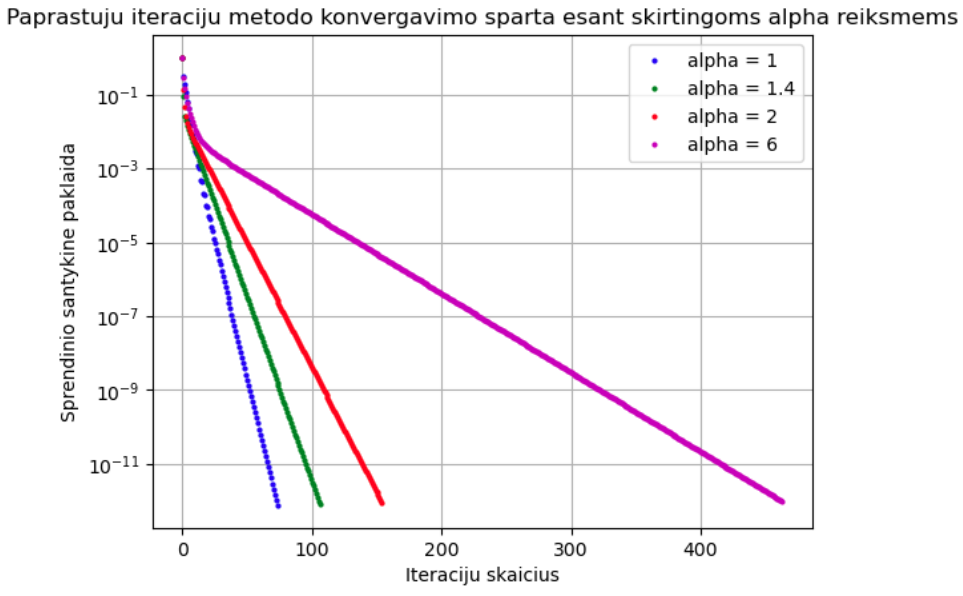
Pradžioje prieš vykdant veiksmus, patikriname, kad mūsų A matricos įstrižainėje nebūtų 0, jeigu yra sukeičiame vietomis.

Mūsų pradinėje matricoje įstrižainėje nėra 0, todėl nieko daryti nereikia.

Taipogi, mums reikia pasirinkti alpha reikšmes, tai nulems mūsų konvergavimo spartą, t.y. sprendinys bus surastas greičiau, jeigu pasirinksime gerą alpha reikšmę (17 pav.).

Parinktos alpha reikšmės:

1, 1.4, 2, 6.



17 pav. PI metodo alpha reikšmės konvergavimo sparta.

Algoritmo principas labai paprastas, pradžioje susikonstruojame matricą, svarbu, kad matrica neturėtų 0 savo įstrižainėje, bei nebūtų singuliari. Konstruojamos naujos matricos, pagal mūsų nurodytą alpha reikšmę ir tikrinama ar juos konverguoja, jeigu konverguoja – artėjame prie sprendinio.

Pasirenkame alpha = 1, nes iš lentelės pavyzdžio (17 pav.), tai optimaliausias koeficientas.

```
Naudojamos alpha reiksmes
[1 1 1 1]
gauti x:
[[ 1.]
 [ 1.]
 [-2.]
 [ 3.]]
```

18 pav. PI metodo x matricos rezultatai.

Šiuos rezultatus palyginame su mūsų išoriniais ištekliais gauto linalg.solve() rezultat ais.

```
x naudojant linalg.solve()
[[ 1.]
 [ 1.]
 [-2.]
 [ 3.]]
```

18 pav. PI metodo linalg.solve() rezultatai.

Įstatome šiuos rezultatus į mūsų lygtis ir patikriname ar sutampa su B pradinės matricos koeficientais.

```
Istatytos x reiksmes i pradine lygciu sistema
[[ 12.]
 [ 10.]
 [-28.]
 [ 16.]]
```

19 pav. Reikšmių rezultatai.

Darome išvadą, kad mūsų PI metodo gauti x matricos rezultatai yra artimi teisingiems.

1.6. Pirmosios dalies PI metodo kodo fragmentas

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

##------------------------------------------------- PAPRASTUJU ITERACIJU -----------
--------------


np.set_printoptions(precision=2, suppress=True, linewidth=120)
#Matrica 8 lygciu sistemos
A=np.matrix([[4,3,-1,1],
             [3,9,-2,-2],
             [-1,-2,11,-1],
             [1,-2,-1,5]]).astype(float)        # koeficientu matrica
b=(np.matrix([12,10,-28,16])).transpose().astype(float)   #laisvuju nariu vektorius-
stulpelis

n = (np.shape(A))[0]


def Sprendiniai_I_Pradine(A, b, x_values, n, singular):
    check = np.zeros((n, 1))
    if singular:
        p = np.symbols('p')
        p_val = random.randint(0, 10)
        print(f"pasirinktas p = {p_val}")
        x_values = x_values.subs(p, p_val)
        print("X:")
        print(x_values)
    for i in range(0, n):
        calc = np.dot(A[i, :n], x_values)
        check[i, 0] = calc.item()

    print("Pradiniai b")
    print(b)
    print("Istatytos x reiksmes i pradine lygciu sistema")
    print(check)
    print("Ar visi lygus?")
    print(np.allclose(b, check, atol=1e-9))

def Paprastuju_Iteraciju(A, b, n, alpha):
    if np.abs(np.linalg.det(A[:, :n])) < 1e-9:  # ar matrica singuliari
        singular = True
    else:
        singular = False

    # istrizaineje negali buti 0
    for i in range(0, n):
        if A[i, i] == 0:
            for j in range(0, n):
                if A[j, i] != 0:
                    A[[i, j]] = A[[j, i]]
                    b[[i, j]] = b[[j, i]]
                    break
    for i in range(n):
        if A[i, i] == 0:
            print("lygtis negali buti sprendziama, nes istrizaineje yra 0")
            return 0, 0
    Atld = np.diag(1. / np.diag(A)).dot(A) - np.diag(alpha)  # istrizaine 1-a, kiti /
is istrizainio
    btld = np.diag(1. / np.diag(A)).dot(b)

    nitmax = 1000
    eps = 1e-12

    x = np.zeros(shape=(n, 1))
    x1 = np.zeros(shape=(n, 1))
    iter_end = 0
    prec_arr = []
    for it in range(0, nitmax):
        x1 = ((btld - Atld.dot(x)).transpose() / alpha).transpose()  # x1 = [a]^-
1({b~}-[A~]{x})
```

```
        prec = np.linalg.norm(x1 - x) / (np.linalg.norm(x) + np.linalg.norm(x1))
        prec_arr.append(prec)
        if prec < eps:
            iter_end = it
            break
        x[:] = x1[:]

    if iter_end == 0 and singular:
        print("Sprendinys nekonvergavo")
        return 0, 0, prec_arr
    elif iter_end != 0 and singular:
        print("Sprendinys konvergavo i viena is galimu sprendiniu")
    return x, iter_end, prec_arr


if __name__ == '__main__':
    Alphas = [np.array([1, 1, 1, 1]), np.array([1.4, 1.4, 1.4, 1.4]), np.array([2, 2,
2, 2]),
              np.array([6, 6, 6, 6])]
    prec_arr=[]
    for alpha in Alphas:
        x, iter_end, prec_temp = Paprastuju_Iteraciju(A, b, n, alpha)
        prec_arr.append(prec_temp)
        if iter_end != 0:
            print("Naudojamos alpha reiksmes")
            print(alpha)
            print("gauti x:")
            print(x)
            print(f"po {iter_end} iteraciju")
            Sprendiniai_I_Pradine(A, b, x, n, False)

            print("x naudojant linalg.solve()")
            solution = np.linalg.solve(A, b)
            print(solution)
    plt.figure()
    plt.title("Paprastuju iteraciju metodo konvergavimo sparta esant skirtingoms alpha
reiksmems")
    colors=['b', 'g', 'r', 'm']
    plt.ylabel("Sprendinio santykine paklaida")
    plt.xlabel("Iteraciju skaicius")
    plt.yscale('log')
    plt.grid()
    for i in range(0,len(prec_arr)):
        plt.plot(prec_arr[i], marker='o', color=colors[i], markersize=2, label=f'alpha
= {Alphas[i][0]}',linestyle=' ', linewidth=1)
    plt.legend()
    plt.show()
```

1.7. Tiesinė lygčių sistema skaidos metodu (6 pav.)

a) Lentelėje 3 duotos tiesinės lygčių sistemos, laisvųjų narių vektoriai ir nurodytas skaidos metodas. Reikia suprogramuoti nurodytą metodą ir juo išspręsti pateiktas lygčių sistemas

10.	$\begin{cases} 6x_1 + x_2 + 3x_3 - 2x_4 = \cdots \\ 6x_1 + 8x_2 + x_3 - x_4 = \cdots \\ 12x_1 - 2x_2 + 4x_3 - x_4 = \cdots \\ 8x_1 + x_2 + x_3 + 5x_4 = \cdots \end{cases}$	$\begin{cases} \ldots = 8 \\ \ldots = 14 \\ \ldots = 13 \\ \ldots = 15 \end{cases}$	$\begin{cases} \ldots = 67 \\ \ldots = 77 \\ \ldots = 126 \\ \ldots = 95 \end{cases}$	$\begin{cases} \ldots = -5.25 \\ \ldots = 0 \\ \ldots = -13.5 \\ \ldots = -7.25 \end{cases}$	LU

20 pav. 3 duota tiesinė lygčių sistema.

$6x_1 + x_2 + 3x_3 - 2x_4 =$	8	67	-5.25
$6x_1 + 8x_2 + x_3 - x_4 =$	14	77	0
$12x_1 - 2x_2 + 4x_3 - x_4 =$	13	126	-13.5
$8x_1 + x_2 + x_3 + 5x_4 =$	15	95	-7.25

Norint, naudoti LU metodą, mūsų matrica privalo būti nesinguliari ir kvadratinė, kitaip negalėsime išskaidyti į trikampių matricų LU sandaugą ir surasti sprendinio matricos.

Pradiniai duomenys:

Pradinė matrica A	Pradinė matrica B1
6, 1, 3, -2	8
6, 8, 1, -1	14
12, -2, 4, -1	13
8, 1, 1, 5	15

Pradinė matrica A	Pradinė matrica B2
6, 1, 3, -2	67
6, 8, 1, -1	77
12, -2, 4, -1	126
8, 1, 1, 5	95

Pradinė matrica A	Pradinė matrica B3
6, 1, 3, -2	-5.25
6, 8, 1, -1	0
12, -2, 4, -1	-13.5
8, 1, 1, 5	-7.25

Gauname rezultatus:

```
Atsakymas (b):
[[ 1.38]
 [ 0.8 ]
 [-0.25]
 [ 0.91]]
Atsakymo patikrinimas:
[[ 8.]
 [14.]
 [13.]
 [15.]]
Liekana:
[[0.]
 [0.]
 [0.]
 [0.]]
Bendra santykinė paklaida:
0.0
```

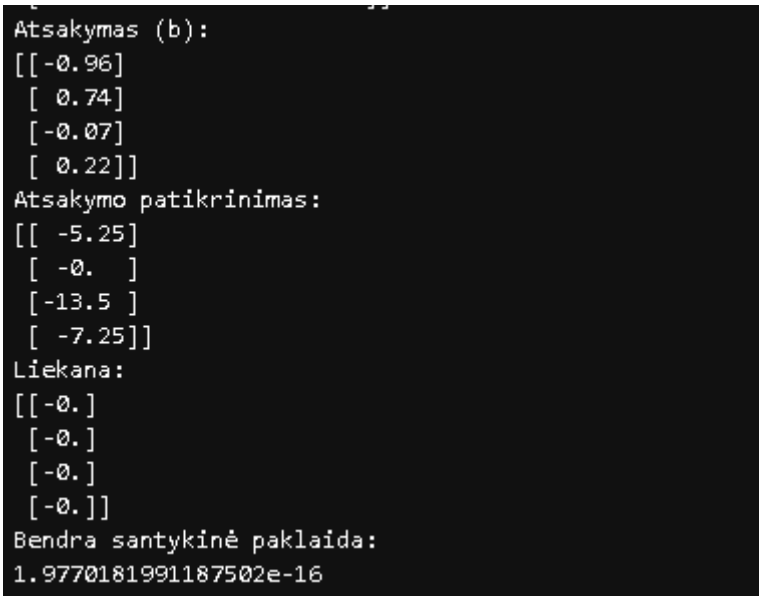
21 pav. B1 rezultatai.

Kadangi mūsų liekanos matricos skaičiai yra maži ir mūsų santykinė paklaida yra gana mažas skaičius, tai galime daryti išvadą, kad mūsų B1 sprendinio matricos rezultatai yra artimi teisingiems.

```
Atsakymas (b):
[[11.08]
 [ 1.41]
 [-0.66]
 [ 1.43]]
Atsakymo patikrinimas:
[[ 67.]
 [ 77.]
 [126.]
 [ 95.]]
Liekana:
[[0.]
 [0.]
 [0.]
 [0.]]
Bendra santykinė paklaida:
0.0
```

22 pav. B2 rezultatai.

Kadangi mūsų liekanos matricos skaičiai yra maži ir mūsų santykinė paklaida yra gana mažas skaičius, tai galime daryti išvadą, kad mūsų B2 sprendinio matricos rezultatai yra artimi teisingiems.



23 pav. B3 rezultatai.

Kadangi mūsų liekanos matricos skaičiai yra maži ir mūsų santykinė paklaida yra gana mažas skaičius, tai galime daryti išvadą, kad mūsų B3 sprendinio matricos rezultatai yra artimi teisingiems.

1.8. Skaidos algoritmo kodo fragmentas:

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


np.set_printoptions(precision=2, suppress=True, linewidth=120)

#---------------------------------------------------- LU SKAIDOS ALGORITMAS -----------
----------------------------------------


# -------------------iseities duomnenys:
A=np.matrix([[6, 1, -3, -2],
             [6, 8, -1, -1],
             [12, -2, 4, -1],
             [8, -1, -1, 5]]).astype(np.float64)        # koeficientu matrica

if np.linalg.det(A):
```

```
    print("Matrica singuliari: Atsakymų yra begalo daug arba sprendinių nėra")

b1=(np.matrix([8, 14, 13, 15])).transpose().astype(np.float64)  #laisvuju nariu
vektorius-stulpelis
b2=(np.matrix([67, 77, 126, 95])).transpose().astype(np.float64)  #laisvuju nariu
vektorius-stulpelis
b3=(np.matrix([-5.25, 0, -13.5, -7.25])).transpose().astype(np.float64)  #laisvuju
nariu vektorius-stulpelis

b_array = [b1, b2, b3]

As=np.matrix(A) # Koeficinetų matricos kopija

n=(np.shape(A))[0] # n - eilutės ilgis matricoje
nb=(np.shape(b1))[1] # nb - stulpelio ilgis matricoje
P=np.arange(0,n)

# Skaičiuojame [L] ir [U] matricas.
for i in range (0,n-1):   # range pradeda 0 ir baigia n-2 (!)
    a=max(abs(A[i:n,i]));  iii=abs(A[i:n,i]).argmax()
    A[[i,i+iii],:]=A[[i+iii,i],:]  # sukeiciamos eilutes
    P[[i,i+iii]]=P[[i+iii,i]]       # Saugome eilučių numerius kai reiks sukeisti
laisvuju nariu vektoriu, nes didžiausias stulpelyje keliauja į viršų
    for j in range (i+1,n):  # range pradeda i+1 ir baigia n-1
        r=A[j,i]/A[i,i] # Suskaičiuojame dauginamaji
        A[j,i:n+nb]=A[j,i:n+nb]-A[i,i:n+nb]*r; # Suskaičiuojame U Matrica, kas yra
šiuo metu gauso
        A[j,i]=r; # Nustatome L Matricos reiksme

# Einame per visus {b} ir skačiuojames atgalinį etapą:
for i in range(len(b_array)):
    print("---------------------------------------------")
    print("matrica: {i}")
    b = b_array[i]
    bs=np.matrix(b) # laisvųjų narių vektoriaus kopija
    b=b[P,:]
    print(f"pertvarkyta laisvųjų narių vektorius (b):\n{b}")
    print(f"L U Matrica:\n {A}")

    # 1-as atvirkstinis etapas, sprendziama  {y} = [L]^(-1) * {b}; b = {y}
    for i in range(1,n) :
        b[i,:]=b[i,:]-A[i,0:i]*b[0:i,:]

    # 2-as atvirkstinis etapas , sprendziama {x} = [U]^(-1) * {y}; b = {x}
    for i in range (n-1,-1,-1) :
        b[i,:]=(b[i,:]-A[i,i+1:n]*b[i+1:n,:])/A[i,i]


    print(f"Atsakymas (b):\n{b}")
    print(f"Atsakymo patikrinimas:\n{As.dot(b)}")
    liekana = As.dot(b)-bs
    print(f"Liekana:\n{liekana}")
    print(f"Bendra santykinė paklaida:\n{np.linalg.norm(liekana)/
np.linalg.norm(bs)}")
```

2. 2 dalis

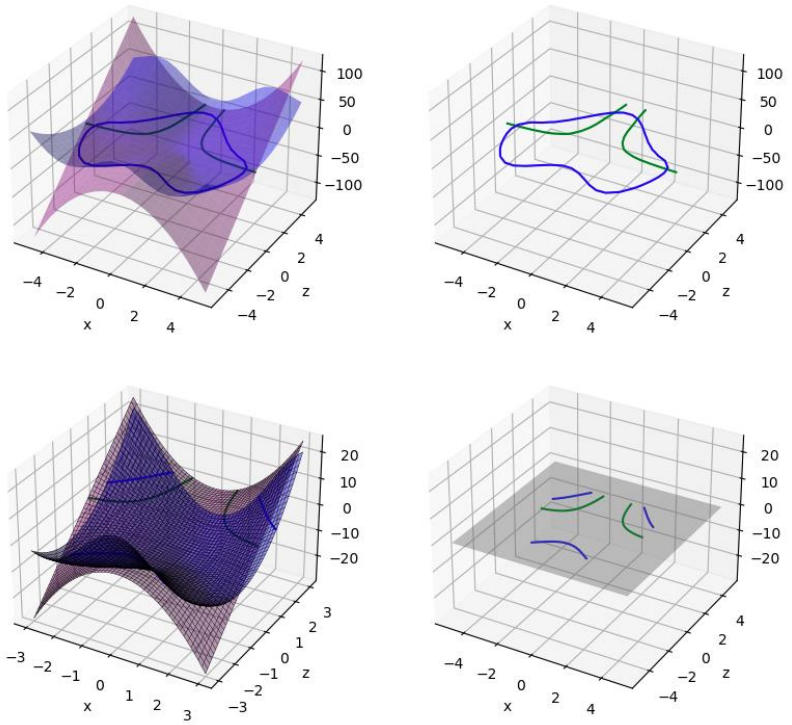
Duota netiesinių lygčių sistema (27 pav.): { $Z1\ (x1, x2\ ) = 0\ Z2\ (x1, x2\ ) = 0$}

10	$\begin{cases} x_1^2 + 2(x_2 - \cos(x_1))^2 - 20 = 0 \\ x_1^2 x_2 - 2 = 0 \end{cases}$	Broideno

24 pav. Antros dalies užduotis.

a. Skirtinguose grafikuose pavaizduokite paviršius $Z1\ (x1, x2\ )$ ir $Z2\ (x1, x2\ )$.

Skirtinguose grafikuose pavaizduojame Z1 ir Z2 paviršius.



25 pav. Antros dalies a) ir b) dalies rezultatai.

b. Užduotyje pateiktą netiesinių lygčių sistemą išspręskite grafiniu būdu.

Užduotį galime išspręsti grafiniu būdu, patikrindami, kur šitie du paviršiai susikerta, susikirtimo taško koordinates laikydami mūsų sprendinių.

Iš grafiko galime atsirinkti šiuos pradinius taškus (artinius).

$$x_1 = -4, \quad x_2 = -0.1$$
$$x_1 = -1, \quad x_2 = 3$$
$$x_1 = 0.5, \quad x_2 = 2.5$$
$$x_1 = 5, \quad x_2 = 0$$

2.1. 2 dalies a) ir b) kodo fragmentas

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits import mplot3d
import math


# #---------------------------------------A DALIS-----------------------------------
------
def LF(x):  # grazina reiksmiu stulpeli
    s=np.array( [x[0]**2+ 2*(x[1]-np.cos(x[0]))**2 - 20, (x[0])**2 * x[1] - 2 ])
    return s
#
#------ Grafika: funkciju LF pavirsiai ------------------------------------------
------------------------
fig1=plt.figure(1,figsize=plt.figaspect(0.5));
ax1 = fig1.add_subplot(1, 2, 1,
projection='3d');ax1.set_xlabel('x');ax1.set_ylabel('y');ax1.set_ylabel('z')
ax2 = fig1.add_subplot(1, 2, 2,
projection='3d');ax2.set_xlabel('x');ax2.set_ylabel('y');ax2.set_ylabel('z')
plt.draw();  #plt.pause(1);
xx=np.linspace(-5,5,20);yy=np.linspace(-5,5,20);
Z=np.zeros(shape=(len(xx),len(yy),2))
X, Y = np.meshgrid(xx, yy);
for i in range (0,len(xx)):
  for j in range (0,len(yy)): Z[i,j,:]=LF([X[i][j],Y[i][j]]).transpose();

surf1 = ax1.plot_surface(X, Y, Z[:,:,0], color='blue', alpha=0.4, linewidth=0.1,
antialiased=True)
CS11 =  ax1.contour(X, Y, Z[:,:,0],[0],colors='b')
surf2 = ax1.plot_surface(X, Y, Z[:,:,1], color='purple', alpha=0.4, linewidth=0.1,
antialiased=True)
CS12 =  ax1.contour(X, Y, Z[:,:,1],[0],colors='g')

CS1 =   ax2.contour(X, Y, Z[:,:,0],[0],colors='b')
CS2 =   ax2.contour(X, Y, Z[:,:,1],[0],colors='g')
plt.show()

# ------------------------------------------------------------------------------------

# ---------------------------------------B DALIS-----------------------------------
-----

def Pavirsius(X,Y,LFF):
    siz=np.shape(X)
    Z=np.zeros(shape=(siz[0],siz[1],2))
    for i in range (0,siz[0]):
        for j in range (0,siz[1]):  Z[i,j,:]=LFF([X[i][j],Y[i][j]]).transpose();
    return Z

#-------- Lygciu sistemos funkcija---------
def LF(x):  # grazina reiksmiu stulpeli
    s=np.array( [x[0]**2+ 2*(x[1]-np.cos(x[0]))**2 - 20, (x[0])**2 * x[1] - 2 ])
    return s
#---------------------------------

fig1=plt.figure(1,figsize=plt.figaspect(0.5));
ax1 = fig1.add_subplot(1, 2, 1,
projection='3d');ax1.set_xlabel('x');ax1.set_ylabel('y');ax1.set_ylabel('z')
ax2 = fig1.add_subplot(1, 2, 2,
projection='3d');ax2.set_xlabel('x');ax2.set_ylabel('y');ax2.set_ylabel('z')
#plt.draw();  #plt.pause(1);
xx=np.linspace(-3,3,50);yy=np.linspace(-3,3,50);
X, Y = np.meshgrid(xx, yy); Z=Pavirsius(X,Y,LF)

surf1 = ax1.plot_surface(X, Y, Z[:,:,0], color='blue', alpha=0.4)
wire1 = ax1.plot_wireframe(X, Y, Z[:,:,0], color='black', alpha=1, linewidth=0.3,
antialiased=True)
surf2 = ax1.plot_surface(X, Y, Z[:,:,1], color='purple', alpha=0.4)
wire2 = ax1.plot_wireframe(X, Y, Z[:,:,1], color='black', alpha=1, linewidth=0.3,
antialiased=True)
CS11 =  ax1.contour(X, Y, Z[:,:,0],[0],colors='b')
CS12 =  ax1.contour(X, Y, Z[:,:,1],[0],colors='g')
CS1 =   ax2.contour(X, Y, Z[:,:,0],[0],colors='b')
```

```
CS2 =   ax2.contour(X, Y, Z[:,:,1],[0],colors='g')

XX=np.linspace(-5,5,2);  YY=XX; XX, YY = np.meshgrid(XX, YY); ZZ=XX*0
zeroplane = ax2.plot_surface(XX, YY, ZZ, color='gray', alpha=0.4, linewidth=0,
antialiased=True)

plt.show()

#-------------------------------------------------------------------------------------
----------
```

c. Nagrinėjamoje srityje sudarykite stačiakampį tinklelį ($x1$, $x2$ poras). Naudodami užduotyje nurodytą metodą apskaičiuokite netiesinių lygčių sistemos sprendinius, kai pradinis artinys įgyja tinklelio koordinačių reikšmes. Tinklelyje vienodai pažymėkite taškus, kuriuos naudojant kaip pradinius artinius gaunamas tas pats sprendinys. Lentelėje pateikite apskaičiuotus skirtingus sistemos sprendinius ir bent po vieną jam atitinkantį pradinį artinį.

Mūsų pradiniai artiniai (sprendžiame Broideno metodu):

$$x_1 = -4, \quad x_2 = -0.1$$
$$x_1 = -1, \quad x_2 = 3$$
$$x_1 = 0.5, \quad x_2 = 2.5$$
$$x_1 = 5, \quad x_2 = 0$$

Pradinis artinys	Iteracijų skaičius	Tikslumas	Rezultatas
$x_1 = -4,$ $x_2 = -0.1$	7	[[2.10525778e-07]]	[[4.44160772 0.10137937]]
$x_1 = -1, \quad x_2 = 3$	6	[[4.14930748e-08]]	[[−0.71851577 3.87398007]]
$x_1 = 0.5, \quad x_2 = 2.5$	8	[[1.65068524e-08]]	[[0.71851577 3.87398007]]
$x_1 = 5, \quad x_2 = 0$	6	[[5.81838783e-07]]	[[4.44160794 0.10137916]]

d. Gautus sprendinius patikrinkite naudodami išorinius išteklius (pvz., standartines Python funkcijas).

Patikrinimui naudojame fsolve() Python funkciją, kuri mums grąžiną rezultatus.

```
Sprendinys:
[[4.44160772 0.10137937]]

Galutinis tikslumas:
[[2.10525778e-07]]

Iteraciju skaičius:
7

Išoriniu ištekliu atsakymas:
[4.44160772 0.10137937]
Broideno metodo suskaičiuota reikšmė yra artima fsolve() rezultatui.
```

26 pav. Pirmojo artinio rezultatai.

```
Sprendinys:
[[-0.71851577  3.87398007]]

Galutinis tikslumas:
[[4.14930748e-08]]

Iteraciju skaičius:
6

Išoriniu ištekliu atsakymas:
[-0.71851577  3.87398007]
Broideno metodo suskaičiuota reikšmė yra artima fsolve() rezultatui.
```

27 pav. Antrojo artinio rezultatai.

```
Sprendinys:
[[0.71851577 3.87398007]]

Galutinis tikslumas:
[[1.65068524e-08]]

Iteraciju skaičius:
8

Išoriniu ištekliu atsakymas:
[0.71851577 3.87398007]
Broideno metodo suskaičiuota reikšmė yra artima fsolve() rezultatui.
```

28 pav. Trečiojo artinio rezultatai.

```
Sprendinys:
[[4.44160794 0.10137916]]

Galutinis tikslumas:
[[5.81838783e-07]]

Iteraciju skaičius:
6

Išoriniu ištekliu atsakymas:
[4.44160772 0.10137937]
Broideno metodo suskaičiuota reikšmė yra artima fsolve() rezultatui.
```

29 pav. Ketvirto artinio rezultatai.

2.2. 2 dalies c) ir d) dalies kodo fragmentas:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits import mplot3d
import math
import time
from scipy.optimize import fsolve
import pandas as pd

def tikslumas(x1,x2,f1,f2,eps):
    if np.isscalar(x1):
            if np.abs(x1+x2) > eps:
                   s=  np.abs(x1-x2)/(np.abs(x1+x2)+np.abs(f1)+np.abs(f2));
            else:  s=  np.abs(x1-x2)+abs(f1)+abs(f2);
    else:
        if (sum(np.abs(x1+x2)) > eps):
               s=  sum(np.abs(x1-x2))/sum(np.abs(x1+x2)+np.abs(f1)+np.abs(f2));
        else:  s=  sum(np.abs(x1-x2)+abs(f1)+abs(f2));
    return s

#-------- Lygciu sistemos funkcija---------
def LF_scipy(x):
    return np.array([x[0]**2 + 2*(x[1] - np.cos(x[0]))**2 - 20, (x[0])**2 * x[1] - 2])


def LF(x):  # grazina reiksmiu stulpeli
    s=np.array( [x[0]**2+ 2*(x[1]-np.cos(x[0]))**2 - 20, (x[0])**2 * x[1] - 2 ])
    s.shape = (2, 1)
    s = np.matrix(s)
    return s
#---------------------------------

# ******************  Programa ***********************************

n = 2  # lygciu skaicius
x = np.matrix(np.zeros(shape=(n, 1)))
#artiniai
#1variantas
#x[0] = 4
#x[1] = -1
#2variantas
#x[0] = -1
#x[1] = 3
#3variantas
#x[0] = 0.5
#x[1] = 2.5
#4variantas
x[0] = 5
x[1] = 0
x0=[x[0], x[1]]
maxiter = 30  # didziausias leistinas iteraciju skaicius
eps = 1e-6    # reikalaujamas tikslumas
iter = 0 #iteraciju skaicius



dx = 0.1   # dx pradiniam Jakobio matricos iverciui
A = np.matrix(np.zeros(shape=(n, n)))
x1 = np.zeros(shape=(n, 1))
for i in range(0, n):
    x1 = np.matrix(x)
    x1[i] += dx
    A[:, i] = (LF(x1) - LF(x)) / dx

ff = LF(x)
print("\nPradine Jakobio matrica:")
print(A)
print("\nPradine funkcijos reiksme:")
print(ff.transpose())


for i in range (1,maxiter):
    iter += 1
    deltax=-np.linalg.solve(A,ff);
    x1=np.matrix(x+deltax);
```

```
    ff1=LF(x1)
    A+=(ff1-ff-A*deltax)*deltax.transpose()/(deltax.transpose()*deltax);
    tiksl=tikslumas(x,x1,ff,ff1,eps) ;
    ff= ff1;
    x=x1;
    if tiksl < eps: break;
    else:
        print("\nx1:")
        print(x1.transpose())



print("\nSprendinys:")
print(x1.transpose())
print("\nGalutinis tikslumas:")
print(tiksl)
print("\nIteraciju skaičius:")
print(iter)
print("\nIšoriniu ištekliu atsakymas:")
calculated_solution_scipy = fsolve(lambda x: LF_scipy(x).flatten(), x0)
print(calculated_solution_scipy)


if np.allclose(x1.A1, calculated_solution_scipy, atol=eps):
    print("Broideno metodo suskaičiuota reikšmė yra artima fsolve() rezultatui.")
else:
    print("Broideno metodo suskaičiuota reikšmė nėra artima fsolve() rezultatui")
```

3. 3 dalis

Pagal pateiktą uždavinio sąlygą (5 lentelė) sudarykite tikslo funkciją ir išspręskite jį vienu iš gradientinių metodų (gradientiniu, greičiausio nusileidimo). Gautą taškų konfigūraciją pavaizduokite programoje, skirtingais ženklais pavaizduokite duotus ir pridėtus (jei sąlygoje tokių yra) taškus. Ataskaitoje pateikite pradinę ir gautą taškų konfigūracijas, taikytos tikslo funkcijos aprašymą, taikyto metodo pavadinimą ir parametrus, iteracijų skaičių, iteracijų pabaigos sąlygas ir tikslo funkcijos priklausomybės nuo iteracijų skaičiaus grafiką.

Uždavinys 7-10 variantams

Miestas išsidėstęs kvadrate, kurio koordinatės ($-10 \leq \boldsymbol{x} \leq 10$, $-10 \leq \boldsymbol{y} \leq 10$). Mieste yra **n** ($\boldsymbol{n} \geq \mathbf{3}$) vieno tinklo parduotuvių, kurių koordinatės yra žinomos (*Koordinatės gali būti generuojamos atsitiktinai, negali būti kelios parduotuvės toje pačioje vietoje*). Planuojama pastatyti dar **m** ($\boldsymbol{m} \geq \mathbf{3}$) šio tinklo parduotuvių. Parduotuvės pastatymo kaina (vietos netinkamumas) vertinama pagal atstumus iki kitų parduotuvių ir poziciją (koordinates). Reikia parinkti naujų parduotuvių vietas (koordinates) taip, kad parduotuvių pastatymo kainų suma būtų kuo mažesnė (naujos parduotuvės gali būti statomos ir už miesto ribos).

Atstumo tarp dviejų parduotuvių, kurių koordinatės $(\boldsymbol{x_1}, \boldsymbol{y_1})$ ir $(\boldsymbol{x_2}, \boldsymbol{y_2})$, kaina apskaičiuojama pagal formulę:

$$C(x_1, y_1, x_2, y_2) = \exp(-0.3 \cdot ((x_1 - x_2)^2 + (y_1 - y_2)^2))$$

Parduotuvės, kurios koordinatės $(\boldsymbol{x_1}, \boldsymbol{y_1})$, vietos kaina apskaičiuojama pagal formulę:

$$C^P(x_1, y_1) = \frac{x_1^4 + y_1^4}{1000} + \frac{\sin(x_1) + \cos(y_1)}{5} + 0.4$$

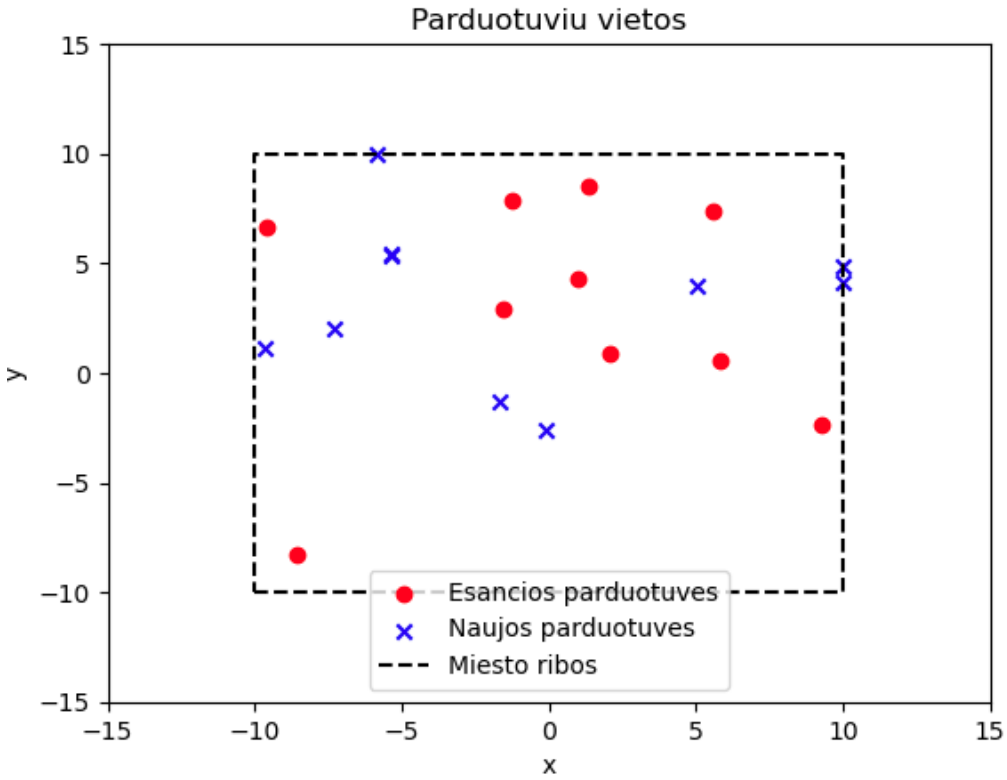
30 pav. Trečios dalies uždavinys.

Pradiniai duomenys ir aprašymai:

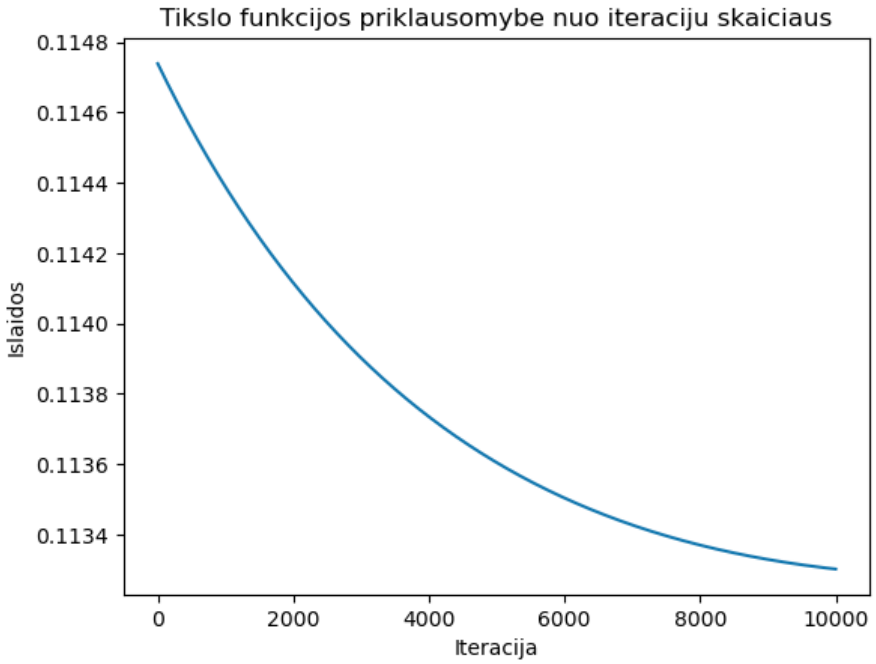
- Tikslo funkcija – skaičiuoja bendras išlaidas, įtraukdama atstumo kainos ir parduotuvės vietos kainos bendras kainas.
- Išlaidos apskaičiuojamos: $cost = distance * localPrice$
- Algoritmui vykstant, peržiūrima, kiekviena nauja ir esama parduotuvės, kai skaičiuojamos išlaidos naujos parduotuvės pastatymui.
- Atstumas – atstumas tarp naujos ir esamos parduotuvės, tam tikra kaina.
- Vietinė kaina – kaina, apskaičiuojama naudojant naujos parduotuvės koordinates.
- Iteracijų pabaigos sąlyga – mažiausios naujos parduotuvės išlaidos.
- Iteracijų skaičius: 10000.
- Miesto ribos: (-10 <= x <= 10, -10<= y <= 10).
- Taikytas metodas: Gradiento greičiausio nusileidimo metodas.
- Žingsnio dydis – gradiento nusileidimo žingsnis, naudojamas atnaujinant parduotuvių koordinates.

Rezultatai:

Pradinė ir gauta taškų konfigūracija, atlikus 10000 iteracijų (30 pav.) ir tikslo funkcijos priklausomybės nuo iteracijų skaičiaus grafikas (31 pav.). Iš grafiko matome, optimizavimo uždavinio rezultatus, mūsų kaina buvo mažinama su vis didėjančiu iteracijų skaičiumi.



31pav. Pradinė ir gauta taškų konfigūracija.



32pav. Tikslo funkcijos priklausomybės nuo iteracijų skaičiaus grafikas.

3.1. Trečios dalies kodo fragmentas:

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

def objective_function(store_locations, new_store_locations):
    cost = 0
    n = len(store_locations)
    m = len(new_store_locations)

    for i in range(m):
        for j in range(n):
            x1, y1 = new_store_locations[i]
            x2, y2 = store_locations[j]
            distance = np.exp(-0.3 * ((x1 - x2) ** 2 + (y1 - y2) ** 2))
            local_price = (x1 ** 4 + y1 ** 4) / 1000 + (np.sin(x1) + np.cos(y1)) / 5 +
0.4
            cost += distance * local_price

    return cost

# Funkcijos gradiento skaiciavimas
def compute_gradient(store_locations, new_store_locations):
    gradient = np.zeros_like(new_store_locations)
    n = len(store_locations)
    m = len(new_store_locations)

    for i in range(m):
        for j in range(2):
            for k in range(n):
                x1, y1 = new_store_locations[i]
                x2, y2 = store_locations[k]
                distance = np.exp(-0.3 * ((x1 - x2) ** 2 + (y1 - y2) ** 2))
                local_price = (x1 ** 4 + y1 ** 4) / 1000 + (np.sin(x1) + np.cos(y1)) /
5 + 0.4

                # Apskaiciuojama i-tosios naujos parduotuves ir k-tosios esamosios
parduotuves gradiento komponente
                gradient[i][j] += -2 * distance * local_price * 0.3 * (x1 - x2) if j
== 0 else -2 * distance * local_price * 0.3 * (y1 - y2)

    return gradient

# Gradiento nuolydzio optimizavimas
def gradient_descent(store_locations, new_store_locations, learning_rate,
num_iterations):
    for iteration in range(num_iterations):
        gradient = compute_gradient(store_locations, new_store_locations)
        new_store_locations -= learning_rate * gradient
        # Naujos parduotuves miesto ribose apribojimas
        new_store_locations = np.clip(new_store_locations, -10, 10)
        cost = objective_function(store_locations, new_store_locations)
        print(f'Iteration {iteration}: Cost = {cost}')
    return new_store_locations

# Atsitiktiniu pradiniu parduotuviu ir nauju parduotuviu generavimas
np.random.seed(0)
n_existing_stores = 10
n_new_stores = 10
store_locations = np.random.uniform(-10, 10, (n_existing_stores, 2))
new_store_locations = np.random.uniform(-10, 10, (n_new_stores, 2))

# Optimizavimo parametrai
learning_rate = 0.01
num_iterations = 10000

# Gradiento nuolydis
new_store_locations = gradient_descent(store_locations, new_store_locations,
learning_rate, num_iterations)

plt.scatter(store_locations[:, 0], store_locations[:, 1], color='red', label='Esancios
parduotuves', marker='o')
plt.scatter(new_store_locations[:, 0], new_store_locations[:, 1], color='blue',
label='Naujos parduotuves', marker='x')
plt.xlim(-15, 15)
plt.ylim(-15, 15)
plt.plot([-10, 10, 10, -10, -10], [-10, -10, 10, 10, -10], 'k--', label='Miesto
ribos')
```

```
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Parduotuviu vietos')
plt.show()

cost_history = []
for iteration in range(num_iterations):
    gradient = compute_gradient(store_locations, new_store_locations)
    new_store_locations -= learning_rate * gradient
    new_store_locations = np.clip(new_store_locations, -10, 10)
    cost = objective_function(store_locations, new_store_locations)
    cost_history.append(cost)
    print(f'Iteracija {iteration}: Islaidos = {cost}')

plt.plot(range(num_iterations), cost_history)
plt.xlabel('Iteracija')
plt.ylabel('Islaidos')
plt.title('Tikslo funkcijos priklausomybe nuo iteraciju skaiciaus')
plt.show()
```

4. Literatūra

1. „Moodle“ aplinkoje esantis modulis „Skaitiniai metodai ir algoritmai“
https://moodle.ktu.edu/course/view.php?id=7639