KAUNO TECHNOLOGIJOS UNIVERSITETAS

Informatikos fakultetas

Skaitiniai metodai ir algoritmai (P170B115)

2 laboratorinis darbas

17 variantas

Dėstytojas:

lekt. Andrius Kriščiūnas

Darbą atliko:

IFF – 8/13 Mykolas Paulauskas

KAUNAS, 2020

Turinys

Užduotis

Tikslas

Laboratoriniame darbe yra skaičiuojamos funkcijų sistemos ir optimizuojami uždaviniai. Tiesinės funkcijos bus sprendžiamos naudojant QR skaidos metodą, netiesinės lygčių sistemos sprendžiamos su greičiausio nusileidimo metodą. Paskutinis uždavinys bus optimizuojamas naudojant funkcijos minimizavimą priešingo gradiento kryptimi

Užduoties variantas: 17

Darbo eiga

Pirma užduotis. Tiesinių lygčių sistemų sprendimas

Duota tiesinių lygčių sistema [A][X] = [B] ir jos sprendimui nurodytas metodas (1 lentelė).

1. Išspręskite tiesinių lygčių sistemą. Jeigu sprendinių be galo daug, raskite bent vieną iš jų. Jeigu sprendinių nėra, pagrįskite, kodėl taip yra.
   Jei metodas paremtas matricos pertvarkymu, pateikite matricų išraiškas kiekviename žingsnyje. Jei metodas iteracinis, grafiškai pavaizduokite, kaip atliekant iteracijas kinta santykinis sprendinio tikslumas esant kelioms skirtingoms konvergavimo daugiklio reikšmėms.
2. Patikrinkite gautus sprendinius ir skaidas, įrašydami juos į pradinę lygčių sistemą.
3. Gautą sprendinį patikrinkite naudodami išorinius išteklius (pvz., standartines MATLAB funkcijas).

pav. 1 pirmoji užduotis

Varianto užduotis:

$$\begin{cases} x_1 + 2x_2 + x_3 = -4 \\ 2x_1 + 5x_2 + 4x_4 = 3 \\ 14x_1 - 8x_2 + 4x_3 + x_4 = 7 \\ 4x_1 + 10x_2 + 8x_4 = 2 \end{cases}$$

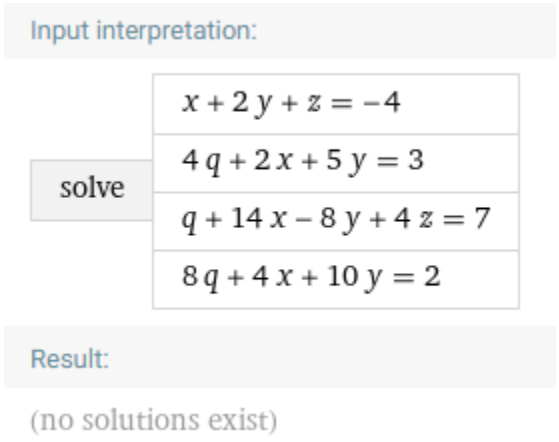
pav. 2 lygčių sistema

Rezultatai:

```
Matrix A
 [[ 1.  2.  1.  0.]
 [ 2.  5.  0.  4.]
 [14. -8.  4.  1.]
 [ 4. 10.  0.  8.]]
Matrix B
 [[-4.]
 [ 3.]
 [ 7.]
 [ 2.]]
Matrix A1
 [[ 1.  2.  1.  0. -4.]
 [ 2.  5.  0.  4.  3.]
 [14. -8.  4.  1.  7.]
 [ 4. 10.  0.  8.  2.]]
Matrix A1
 [[ 1.47309199e+01 -4.07306540e+00  3.86941213e+00  3.66575886e+00
   7.33151772e+00]
 [-2.49800181e-16  5.88458245e+00 -4.17949002e-01  3.46605779e+00
   1.34948892e+00]
 [-4.44089210e-15 -1.80792282e+00  1.07435698e+00 -2.73759548e+00
  -4.55357759e+00]
 [-4.44089210e-16  1.17691649e+01 -8.35898005e-01  6.93211558e+00
  -1.30102217e+00]]
Matrix A1
 [[ 1.47309199e+01 -4.07306540e+00  3.86941213e+00  3.66575886e+00
   7.33151772e+00]
 [ 1.00306164e-16  1.32819478e+01 -1.07210415e+00  8.05084291e+00
   6.48813836e-02]
 [-4.35532579e-15  1.84993066e-16  9.14480872e-01 -1.61706983e+00
  -4.86753684e+00]
 [-1.00110635e-15  1.49632756e-15  2.04858990e-01 -3.62250652e-01
   7.42780875e-01]]
Matrix A1
 [[ 1.47309199e+01 -4.07306540e+00  3.86941213e+00  3.66575886e+00
   7.33151772e+00]
 [ 1.00306164e-16  1.32819478e+01 -1.07210415e+00  8.05084291e+00
   6.48813836e-02]
 [-4.46883210e-15  5.07614410e-16  9.37145917e-01 -1.65714827e+00
  -4.58744355e+00]
 [ 2.48253415e-17 -1.41969934e-15  4.79200829e-16 -2.13016199e-15
  -1.78885438e+00]]
Sprendiniu aibe tuscia, nes 0 * X != -1.7888543819998333
```

pav. 3 Programos rezultatai

Sprendinius tikriname naudodami Wolfram Alpha.



pav. 4 Sprendinių tikrinimas

Programos kodas:

```
import numpy as np


def task1(A, B):
    print("Matrix A\n", A)
    print("Matrix B\n", B)

    n = (np.shape(A))[0]
    m = (np.shape(B))[1]

    A1 = np.hstack((A, B))
    print("Matrix A1\n", A1)

    Q = np.identity(n)
    for i in range(0, n - 1):
        z = np.vstack(A1[i:n, i])
        zMirror = np.zeros(np.shape(z))
        zMirror[0] = np.sign(z[0]) * np.linalg.norm(z)

        omega = (z - zMirror) / np.linalg.norm(z - zMirror)

        iterative_Q = np.identity(n - i) - 2 * omega * omega.transpose()
        A1[i:n, :] = iterative_Q.dot(A1[i:n, :])
        print("Matrix A1\n", A1)

        Q = Q.dot(
```

```
            np.vstack(
                (
                    np.hstack((np.identity(i), np.zeros(shape=(i, n - i)))),
                    np.hstack((np.zeros(shape=(n - i, i)), iterative_Q))
                )
            )
        )

    R = Q.transpose() * B
    x = np.zeros(shape=(n, m))
    eps = 1e-10

    for i in range(n - 1, -1, -1):
        if np.abs(A1[i, i]) < eps and np.abs(A1[i, n]) < eps: #Tikriname singulia
ruma
            x[i] = 1
            print("Kintamasis x[{}] gali buti bet koks skaicius. Tegul x[{}] = 1"
.format(i, i))
        elif np.abs(A1[i, i]) < eps and np.abs(A1[i, n]) > eps:
            print("Sprendiniu aibe tuscia, nes 0 * X != {}".format(A1[i, n]))
            return
        else:
            x[i, :] = (R[i, :] - A1[i, i + 1:n] * x[i + 1:n, :]) / A1[i, i]

    print("A\n", A)
    print("x\n", x)
    print("B\n", B)

    print("Tikrinimas:\n", A * x - B)

#Duoda uzduoties, kai nera sprendiniu
A_nedalinta = [
    [ 1,  2, 1, 0],
    [ 2,  5, 0, 4],
    [14, -8, 4, 1],
    [ 4, 10, 0, 8]
    ]

B_nedalinta = [
    -4,
    3,
    7,
    2
    ]
```

```
#Lygtis su vienu sprendiniu
# A_nedalinta = [
#     [ 2,  1, -1, 2],
#     [ 4,  5, -3, 6],
#     [-2, 5, -2, 6],
#     [ 4, 11, -4, 8]
# ]

# B_nedalinta = [
#      5,
#      9,
#      4,
#      2
# ]

# lygciu sistema, kurioje yra daug sprendiniu
# A = [
#   [2, 5, 1, 2],
#   [-2, 0, 3, 5],
#   [1, 0, -1, 1],
#   [0, 5, 4, 7]
# ]
# B = [
#   14,
#   10,
#   4,
#   24
# ]

A = np.matrix(A_nedalinta).astype(np.float)
B = np.matrix(B_nedalinta).transpose().astype(np.float)

task1(A, B)
```

Antra užduotis. Netiesinių lygčių sistemų sprendimas. Pirma dalis

1. Duota netiesinių lygčių sistema (2 lentelė. I lygčių sistema):

$$\begin{cases} Z_1(x_1, x_2) = 0 \\ Z_2(x_1, x_2) = 0 \end{cases}$$

   a. Skirtinguose grafikuose pavaizduokite paviršius $Z_1(x_1, x_2)$ ir $Z_2(x_1, x_2)$.
   b. Užduotyje pateiktą netiesinių lygčių sistemą išspręskite grafiniu būdu.
   c. Užduotyje pateiktą netiesinių lygčių sistemą išspręskite naudodami užduotyje nurodytą metodą su laisvai pasirinktu pradiniu artiniu (išbandykite bent keturis pradinius artinius). Nurodykite iteracijų pabaigos sąlygas. Lentelėje pateikite pradinį artinį, tikslumą, iteracijų skaičių.
   d. Gautus sprendinius patikrinkite naudodami išorinius išteklius (pvz., standartines MATLAB funkcijas).

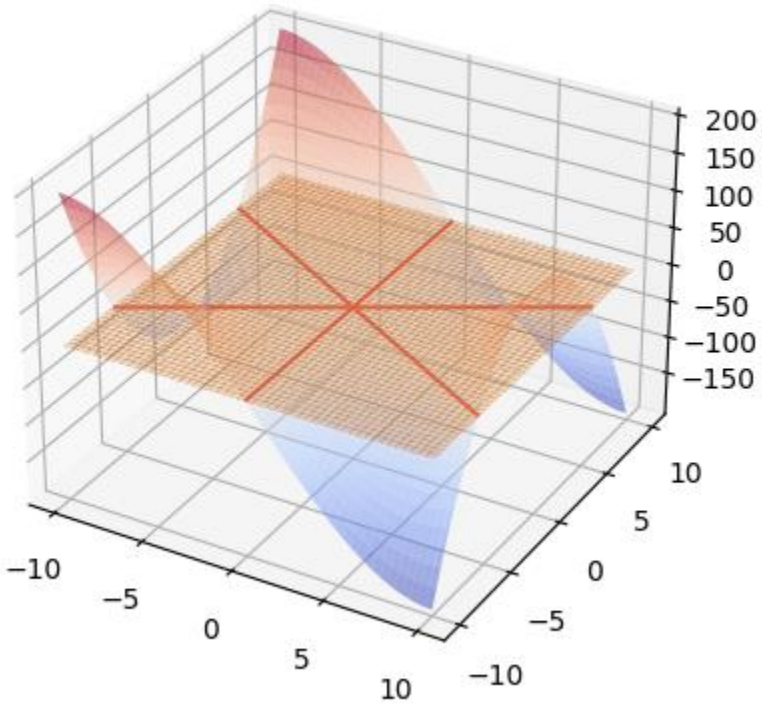
2. Duota netiesinių lygčių sistema (2 lentelė. II lygčių sistema):

$$\begin{cases} Z_1(x_1, x_2, x_3, x_4) = 0 \\ Z_2(x_1, x_2, x_3, x_4) = 0 \\ Z_3(x_1, x_2, x_3, x_4) = 0 \\ Z_4(x_1, x_2, x_3, x_4) = 0 \end{cases}$$

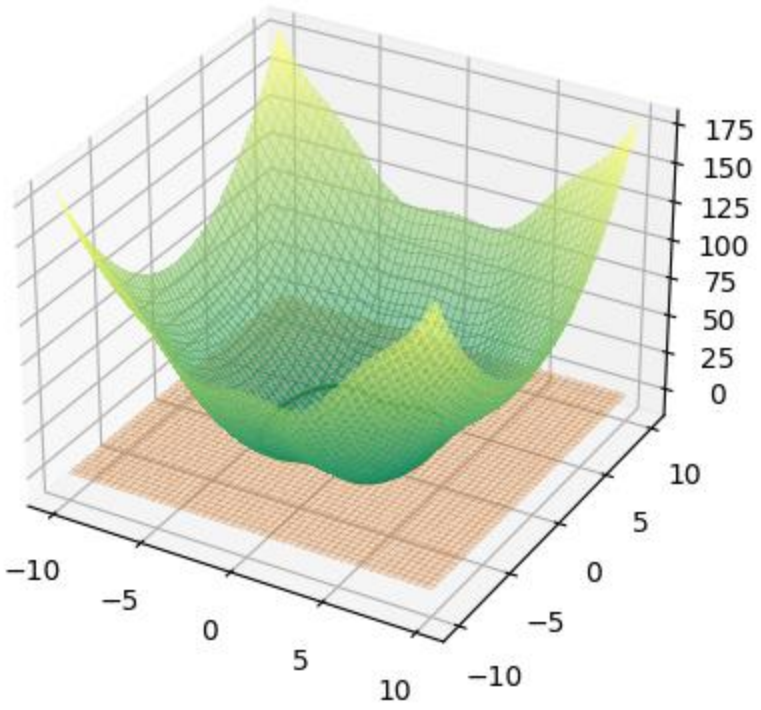
   a. Užduotyje nurodytu metodu išspręskite netiesinių lygčių sistemą su laisvai pasirinktu pradiniu artiniu.
   b. Gautą sprendinį patikrinkite naudodami išorinius išteklius (pvz., standartines MATLAB funkcijas).

pav. 5 antroji užduotis

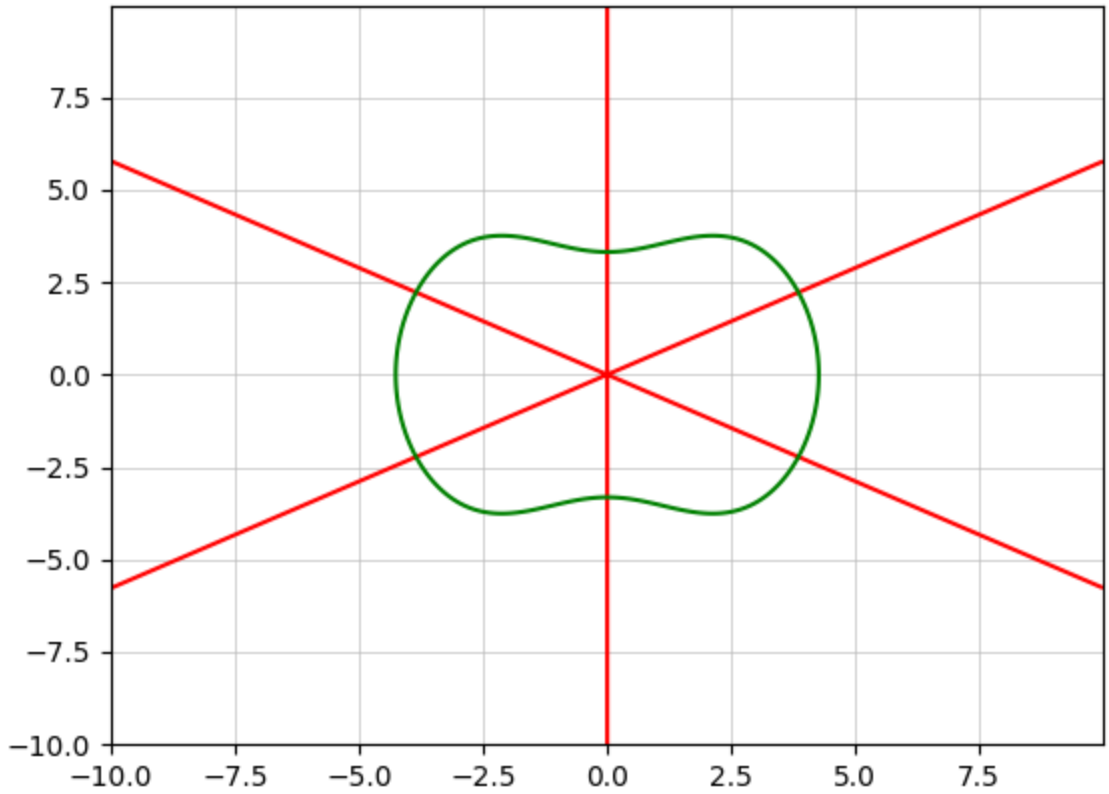
Grafinis sprendimas



pav. 6 Netiesinė sistema, pirma funkcija



pav. 7 Netiesinė sistema, antra funkcija



pav. 8 Netiesinė sistema, grafinis sprendimas

Programos kodas:

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

X = np.arange(-10, 10, 0.01)
Y = np.arange(-10, 10, 0.01)
XX, YY = np.meshgrid(X, Y)

Z1 = 0.1 * XX**3 - 0.3 * XX * YY**2
Z2 = XX**2 + YY**2 + 5 * np.cos(XX) - 16

figure = plt.figure()
aXis = figure.gca(projection='3d')
surf = aXis.plot_surface(XX, YY, Z1, cmap=cm.coolwarm, alpha=0.5)
surfZ = aXis.plot_surface(XX, YY, np.zeros(np.shape(Z1)), antialiased=False, alph
a=0.2)
cp = aXis.contour(X, Y, Z1, levels=0, colors='red')
```

```
plt.show()

figure = plt.figure()
aXis = figure.gca(projection='3d')
surf = aXis.plot_surface(XX, YY, Z2, cmap=cm.summer, antialiased=False, alpha=0.5
)
surfZ = aXis.plot_surface(XX, YY, np.zeros(np.shape(Z1)), antialiased=False, alph
a=0.2)
cp = aXis.contour(X, Y, Z2, levels=0, colors='green')
plt.show()

figure = plt.figure()
aXis = figure.gca()
aXis.grid(color='#C0C0C0', linestyle='-', linewidth=0.5)
cp = aXis.contour(X, Y, Z1, levels=0, colors='red')
cp = aXis.contour(X, Y, Z2, levels=0, colors='green')
plt.show()
```

Dvieju lygčių sistemą (pirma lentelė)

$$\begin{cases} 0.1x_1^3 - 0.3x_1x_2^2 = 0 \\ x_1^2 + x_2^2 + 5\cos(x_1) - 16 = 0 \end{cases}$$

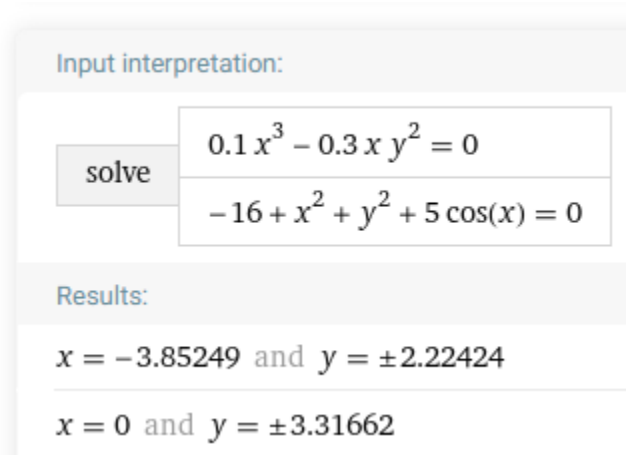
Parametrai:

iterationMax	step	eps
50	0.01	1e-8

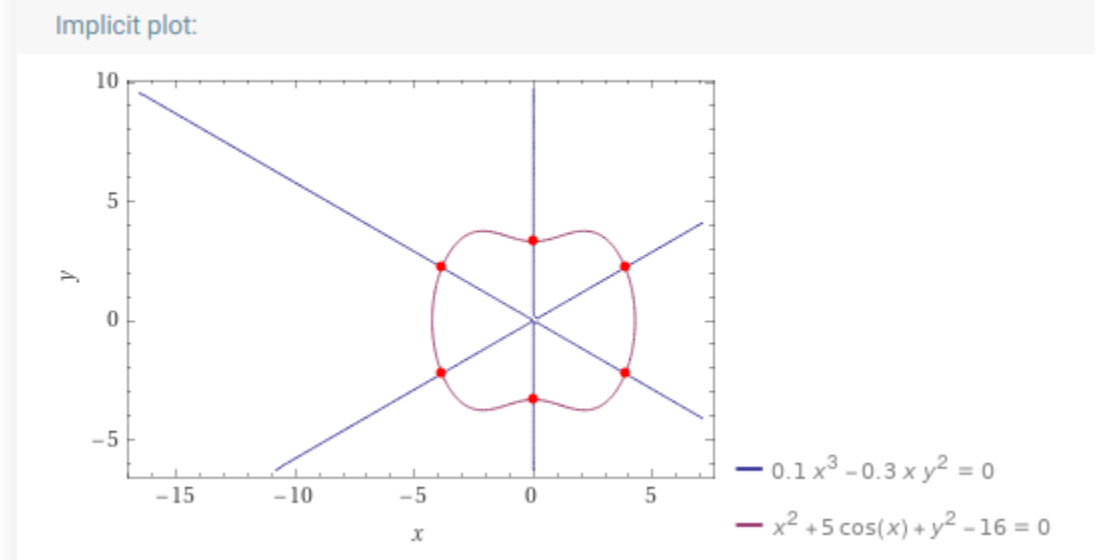
Rezultatai:

Iteracijos	Pradinis artinys	Sprendinys	Tikslumas
18	(-4, 2)	(-3.8524967480412955, 2.2242299136460395)	3.71147e-09
11	(-3.9, -2.1)	(-3.8524888099897403, -2.224215491998147)	8.67215e-09
13	(0.1, 3)	(-1.1685774208408011e-05, 3.316624049155811)	7.55641e-10
13	(-0.1, -3)	(1.1685774208408011e-05, -3.316624049155811)	7.55641e-10

Sprendinius tikriname naudodami Wolfram Alpha.



pav. 9 Wolfram Alpha sprendimas



pav. 10 Wolfram Alpha grafikas

Programos kodas:

```
import numpy as np


def f1(x1, x2):
    return 0.1 * x1 ** 3 - 0.3 * x1 * x2 ** 2


def f2(x1, x2):
    return x1 ** 2 + x2 ** 2 + 5 * np.cos(x1) - 16


def f(x1, x2):
    return np.array([[f1(x1, x2)], [f2(x1, x2)]])


# jakobio matrica
def jacobianMatrix(x1, x2):
    jacobian = np.zeros(shape=(2, 2))
    jacobian[0, 0] = 0.3 * x1 ** 2 - 0.3 * x2 ** 2
    jacobian[0, 1] = -0.6 * x1 * x2
    jacobian[1, 0] = 2 * x1 - 5 * np.sin(x1)
    jacobian[1, 1] = 2 * x2

    return jacobian


# tikslo funkcija
def target(x1, x2):
    return np.dot(np.transpose(f(x1, x2)), f(x1, x2)) / 2


# gradientas
def gradient(x1, x2):
    return np.matmul(np.transpose(f(x1, x2)), jacobianMatrix(x1, x2))

#Greiciausio nusileidimo funkcija
def descent(x1, x2, iterationMax, step, eps):
    step0 = step
    for i in range(iterationMax):
        g = gradient(x1, x2)
        value = target(x1, x2)
        for j in range(30):
            n = np.linalg.norm(g)
```

```
            diferentialx = g / n * step
            x1 = (x1 - diferentialx[0][0])
            x2 = (x2 - diferentialx[0][1])

            valueNext = target(x1, x2)
            if valueNext > value: #Jeigu sekanti funkcijos verte didesne negu esa
ma, griztama atgal ir mazinamas zingsnis
                x1 = x1 + diferentialx[0][0]
                x2 = x2 + diferentialx[0][1]
                step = step / 10

            else:
                value = valueNext
        step = step0
        precision = np.linalg.norm(value)
        print("i:%g, x1=%g, x2=%g, precision=%g" % (i + 1, x1, x2, precision))
        if precision < eps:
            return x1, x2

    return "tikslumas nepasiektas"

# print(descent(-4, 2, 50, 0.01, 1e-8))
# print(descent(-3.9, -2.1, 50, 0.01, 1e-8))
# print(descent(0.1, 3, 50, 0.01, 1e-8))
print(descent(-0.1, -3, 50, 0.01, 1e-8))
```

Keturių lygčių sistema (antra lentelė)

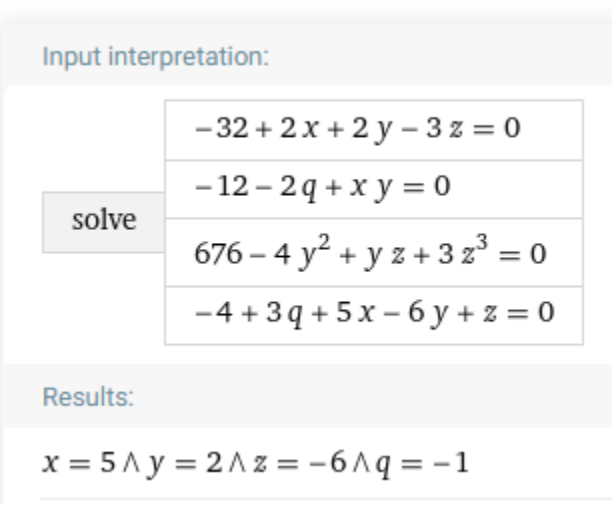
$$\begin{cases} 2x_1 + 2x_2 - 3x_3 - 32 = 0 \\ x_1x_2 - 2x_4 - 12 = 0 \\ -4x_2^2 + x_2x_3 + 3x_3^3 + 676 = 0 \\ 5x_1 - 6x_2 + x_3 + 3x_4 - 4 = 0 \end{cases}$$

iterationMax	step	eps
10000	0.1	1e-5

Iteracijos	Pradinis artinys	Sprendinys	Tikslumas
6493	(4.99, 1.99, -5.99, -0.99)	(4.998680805006482, 2.0023318267606522,	9.99838e-06

		-5.999847520560582, -0.9937586510871118)	

Sprendinius tikriname naudodami Wolfram Alpha.



pav. 11 Wolfram Alpha sprendimas

Programos kodas:

```
import numpy as np


def f1(x1, x2, x3):
    return 2 * x1 + 2 * x2 - 3 * x3 - 32


def f2(x1, x2, x4):
    return x1 * x2 - 2 * x4 - 12


def f3(x2, x3):
    return -4 * x2 ** 2 + x2 * x3 + 3 * x3 ** 3 + 676


def f4(x1, x2, x3, x4):
    return 5 * x1 - 6 * x2 + x3 + 3 * x4 - 4


def f(x1, x2, x3, x4):
    return np.array([[f1(x1, x2, x3)], [f2(x1, x2, x4)], [f3(x2, x3)], [f4(x1, x2
, x3, x4)]])
```

```
# jakobio matrica
def jacobianMatrix(x1, x2, x3, x4):
    jacobian = np.zeros(shape=(4, 4))

    jacobian[0, 0] = 2
    jacobian[0, 1] = 2
    jacobian[0, 2] = -3
    jacobian[0, 3] = 0

    jacobian[1, 0] = x2
    jacobian[1, 1] = x1
    jacobian[1, 2] = 0
    jacobian[1, 3] = -2

    jacobian[2, 0] = 0
    jacobian[2, 1] = -8 * x2 + x3
    jacobian[2, 2] = x2 + 9 * x3**2
    jacobian[2, 3] = 0

    jacobian[3, 0] = 5
    jacobian[3, 1] = -6
    jacobian[3, 2] = 1
    jacobian[3, 3] = 3

    return jacobian


# tikslo funkcija
def target(x1, x2, x3, x4):
    return np.dot(np.transpose(f(x1, x2, x3, x4)), f(x1, x2, x3, x4)) / 2


# gradientas
def gradient(x1, x2, x3, x4):
    return np.matmul(np.transpose(f(x1, x2, x3, x4)), jacobianMatrix(x1, x2, x3,
x4))

#Greiciausio nusileidimo funkcija
def descent(x1, x2, x3, x4, iterationMax, step, eps):
    step0 = step
    for i in range(iterationMax):
        g = gradient(x1, x2, x3, x4)
        value = target(x1, x2, x3, x4)
```

```
        for j in range(50):
            n = np.linalg.norm(g)
            diferentialx = g / n * step
            x1 = x1 - diferentialx[0][0]
            x2 = x2 - diferentialx[0][1]
            x3 = x3 - diferentialx[0][2]
            x4 = x4 - diferentialx[0][3]
            valueNext = target(x1, x2, x3, x4)

            #Jeigu sekanti funkcijos verte didesne negu esama, griztama atgal ir
mazinamas zingsnis
            if valueNext > value:
                x1 = x1 + diferentialx[0][0]
                x2 = x2 + diferentialx[0][1]
                x3 = x3 + diferentialx[0][2]
                x4 = x4 + diferentialx[0][3]
                step = step / 10
            else:
                value = valueNext
        step = step0
        precision = np.linalg.norm(value)
        print("i:%g, x1=%g, x2=%g, x3=%g, x4=%g, prec=%g" % (i + 1, x1, x2, x3, x
4, precision))
        if (precision < eps):
            return x1, x2, x3, x4
    return "tikslumas nepasiektas"


print(descent(4.99, 1.99, -5.99, -0.99, 10000, 0.1, 1e-5))
```

Trečia užduotis. Optimizavimas

Pagal pateiktą uždavinio sąlygą (3 lentelė) sudarykite tikslo funkciją ir išspręskite jį vienu iš gradientinių metodų (gradientiniu, greičiausio nusileidimo, kvazi-gradientiniu, ar pan.). 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.

Duotos $\mathbf{n}$ $(3 \leq \boldsymbol{n})$ taškų koordinatės $(-10 \leq \boldsymbol{x} \leq 10,\ -10 \leq \boldsymbol{y} \leq 10)$. *(Koordinatės gali būti generuojamos atsitiktiniu būdu)*. Srityje $(-10 \leq \boldsymbol{x} \leq 10, -10 \leq \boldsymbol{y} \leq 10)$ reikia padėti papildomų $\mathbf{m}$ $(3 \leq \boldsymbol{m})$ taškų taip, kad jų atstumai nuo visų kitų taškų (įskaitant ir papildomus) būtų kuo artimesni vidutiniam atstumui, o atstumas nuo koordinačių pradžios būtų kuo artimesnis nurodytai reikšmei $\mathbf{S}$ $(1 \leq \boldsymbol{S})$.

Tikslo funkcija:

$$\min_{x_{1:m},y_{1:m}} \Psi(x,y) = \sum_{i=1}^{m}\sum_{j=1}^{n}((x_i - x_j)^2 + (y_i - y_j)^2 - d)^2 + \sum_{i=1}^{m}\sum_{j=i+1}^{m}((x_i - x_j)^2 + (y_i - y_j)^2 - d)^2 + \sum_{i=1}^{m}(x^2 + y^2 - S)^2$$

Taikytas metodas: minimizavimas priešinga gradientui kryptimi.

Parametrai: pradinis žingsnis yra 0.1, iteracijų skaičius yra 1000, h yra 0.0001, s yra 1.

Pabaigos sąlyga: ieškoma kol tenkinamas pasirinktas tikslumas arba jeigu pasiektas iteracijų limitas, nutraukiamas metodas.

Pradiniai duomenys:

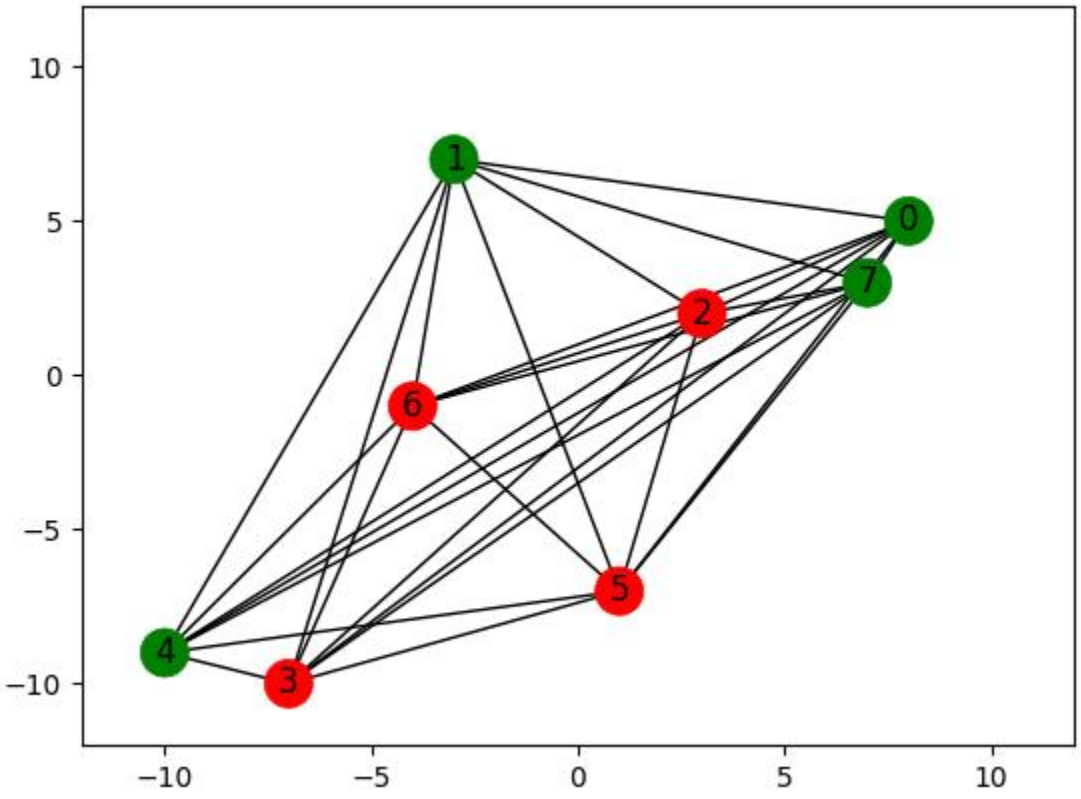
```
x:
 [  8  -3   3  -7 -10   1  -4   7]
y:
 [  5   7   2 -10  -9  -7  -1   3]
```

pav. 12 Pradiniai duomenys

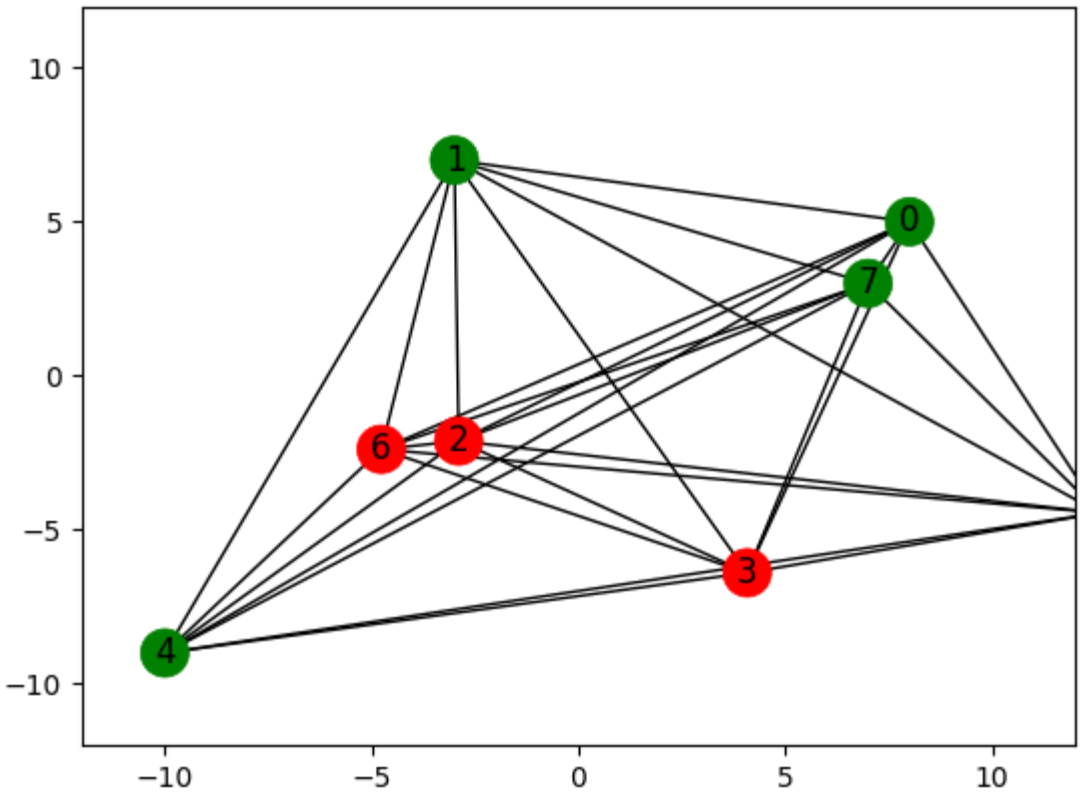
Rezultatai:

```
Rezultatas x:
 [  8.          -3.          -2.89575202   4.07793093 -10.
  12.60407072  -4.76788446   7.         ]
Rezultatas y:
 [ 5.          7.         -2.11835178 -6.39710416 -9.         -4.45145809
 -2.38599007  3.        ]
```

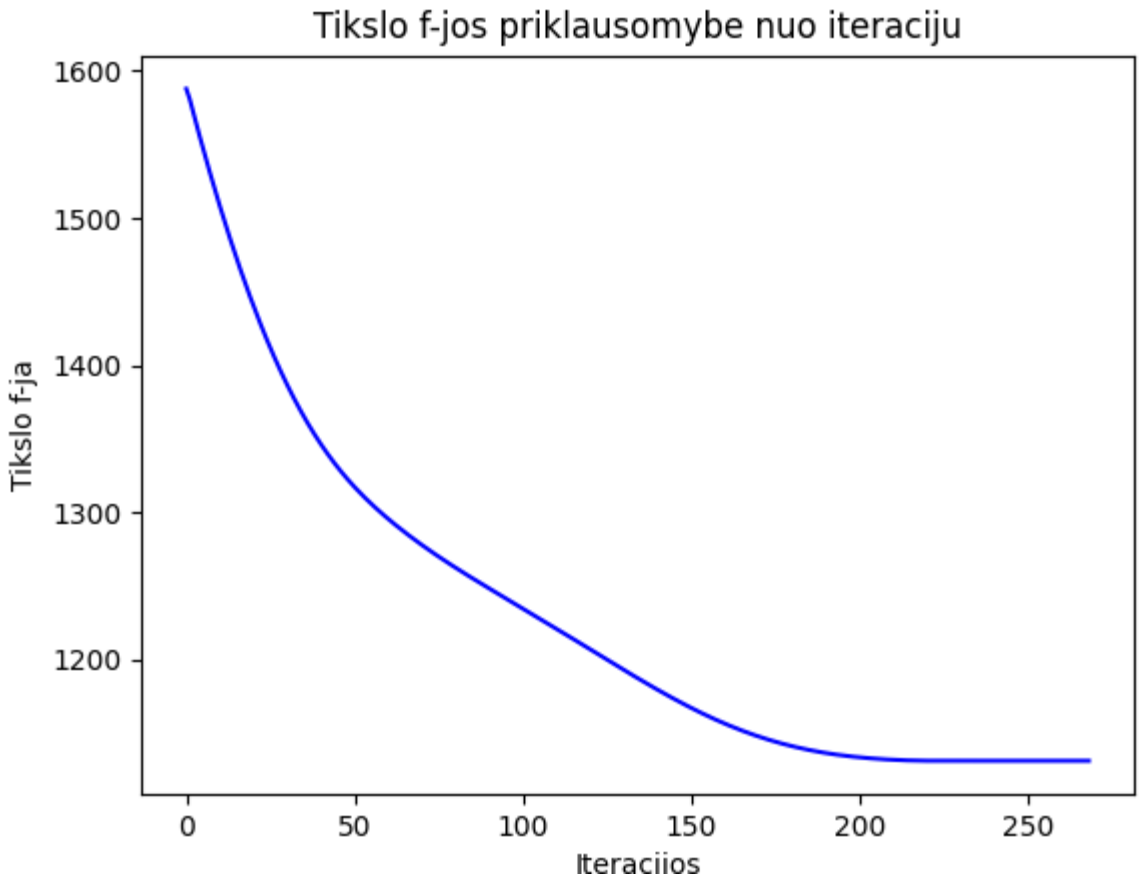
pav. 13 Rezultatai



pav. 14 Pradinė padėtis



pav. 15 Rezultatai



pav. 16 Tikslo funkcijos priklausomybė nuo iteracijų

Programos kodas:

```
import numpy as np
import random
from matplotlib import pyplot as plt
import networkx as nx

#vidutinis atstumas visu tasku
def averageDistance(x, y):
    n = len(x)
    distance = 0

    for i in range(0, n - 1):
        for j in range(i + 1, n):
            distance += np.sqrt(((x[j] - x[i]) ** 2 + (y[j] - y[i]) ** 2))

    return distance / (n * (n - 1) / 2)

#Tikslo funkcija, kurios reiksme turesim minimizuoti
```

```
def targetFunction(x, y, identitie, d, s):
    target = 0.0
    dynamicPoints = []
    n = len(x)

    for i in range(0, n):
        if identitie[i] == 0:
            dynamicPoints.append([x[i], y[i]])

    m = len(dynamicPoints)

    for i in range(0, m):
        for j in range(0, n):
            if i + m != j:
                target += (np.sqrt((x[j] - x[i]) ** 2 + (y[j] - y[i]) ** 2) - d)
** 2
        target += (np.sqrt((dynamicPoints[i][0]) ** 2 + (dynamicPoints[i][1]) **
2) - s) ** 2

    return target

#Gradiento funkcija
def gradient(x, y, identify, d, s):
    h = 0.0001
    n = x.shape[0]
    xGradient = np.zeros(n)
    yGradient = np.zeros(n)
    value = targetFunction(x, y, identify, d, s)

    for i in range(0, n):
        xxx = np.array(x, copy=True)
        xxx[i] += h
        valueNextX = targetFunction(xxx, y, identify, d, s)
        xGradient[i] = (valueNextX - value) / h

    for i in range(0, n):
        yyy = np.array(y, copy=True)
        yyy[i] += h
        valueNextY = targetFunction(x, yyy, identify, d, s)
        yGradient[i] = (valueNextY - value) / h

    s = ([xGradient, yGradient])

    return s
```

```
def drawGraph(x, y, identify):
    n = x.shape[0]
    pos = {}

    for i in range(0, n):
        pos[i] = ([x[i], y[i]])

    graph = nx.complete_graph(x.shape[0])
    color = ["" for i in range(0, x.shape[0])]

    for i in range(0, x.shape[0]):
        if identify[i] == 0:
            color[i] = 'Green'
        else:
            color[i] = 'Red'

    nx.draw_networkx(graph, pos, linewidths=0.5, node_color=color)
    plt.xlim(-12, 12)
    plt.ylim(-12, 12)
    plt.tick_params(left=True, bottom=True, labelleft=True, labelbottom=True)
    plt.show()


def main():
    random.seed(69666)
    n = 8
    x = np.array(random.sample(range(-10, 10), n))
    print("x:\n", x)
    y = np.array(random.sample(range(-10, 10), n))
    print("y:\n", y)
    identify = np.array([0, 0, 1, 1, 0, 1, 1, 0])
    s = 1

    d = averageDistance(x, y)
    print("vidutinis atstumas:", d)

    f = targetFunction(x, y, identify, d, s)

    step0 = 0.1
    step = step0
    maxIterations = 1000

    targetValues = []
    iterations = []
```

```
    drawGraph(x, y, identify)

    for i in range(0, maxIterations):
        grad = gradient(x, y, identify, d, s) * identify
        fff_initial = targetFunction(x, y, identify, d, s)
        derivative = grad / np.linalg.norm(grad) * step
        x = x - derivative[0].transpose()
        y = y - derivative[1].transpose()
        fff_after = targetFunction(x, y, identify, d, s)

        if fff_after > fff_initial:
            x = x + derivative[0].transpose()
            y = y + derivative[1].transpose()
            step /= 2

        precision = np.abs(fff_initial - fff_after) / (np.abs(fff_initial) + np.a
bs(fff_after))
        # print("f = {}".format(fff_after))
        if precision < 1e-16:
            print("Rezultatas x:\n", x)
            print("Rezultatas y:\n", y)
            break
        elif i == maxIterations:
            print("Tikslumas nepasiektas. Paskutinis artinys x = {}".format(x))
            break

        targetValues.append(fff_initial)
        iterations.append(i)

    drawGraph(x, y, identify)

    d = averageDistance(x, y)
    print("vidutinis atstumas:", d)

    plt.plot(iterations, targetValues, 'b-')
    plt.title("Tikslo f-jos priklausomybe nuo iteraciju")
    plt.xlabel('Iteracijos')
    plt.ylabel('Tikslo f-ja')
    plt.show()



main()
```