**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 3](#_Toc57590254)

[Tikslas 3](#_Toc57590255)

[Darbo eiga 3](#_Toc57590256)

[Pirma užduotis. Tiesinių lygčių sistemų sprendimas 3](#_Toc57590257)

[Antra užduotis. Netiesinių lygčių sistemų sprendimas. Pirma dalis 8](#_Toc57590258)

[Grafinis sprendimas 9](#_Toc57590259)

[Dvieju lygčių sistemą (pirma lentelė) 12](#_Toc57590260)

[Keturių lygčių sistema (antra lentelė) 15](#_Toc57590261)

[Trečia užduotis. Optimizavimas 18](#_Toc57590262)

# Užduotis

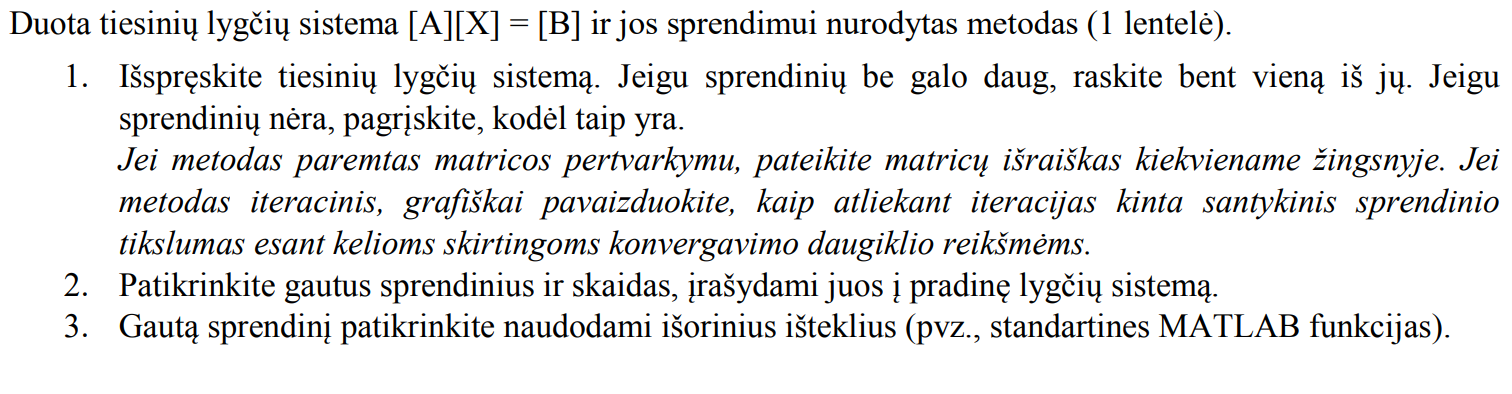
## Tikslas

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

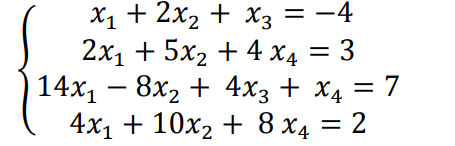
# Darbo eiga

## Pirma užduotis. Tiesinių lygčių sistemų sprendimas



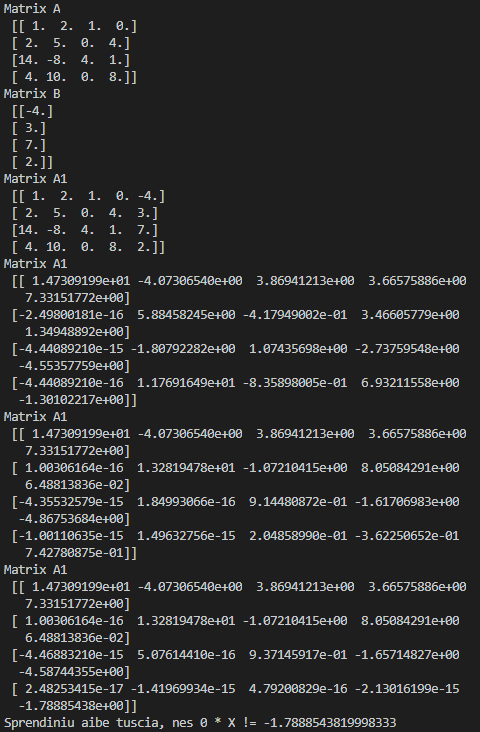
pav. 1 pirmoji užduotis

Varianto užduotis:



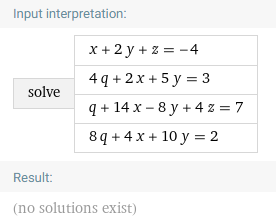
pav. 2 lygčių sistema

Rezultatai:



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 singuliaruma

            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

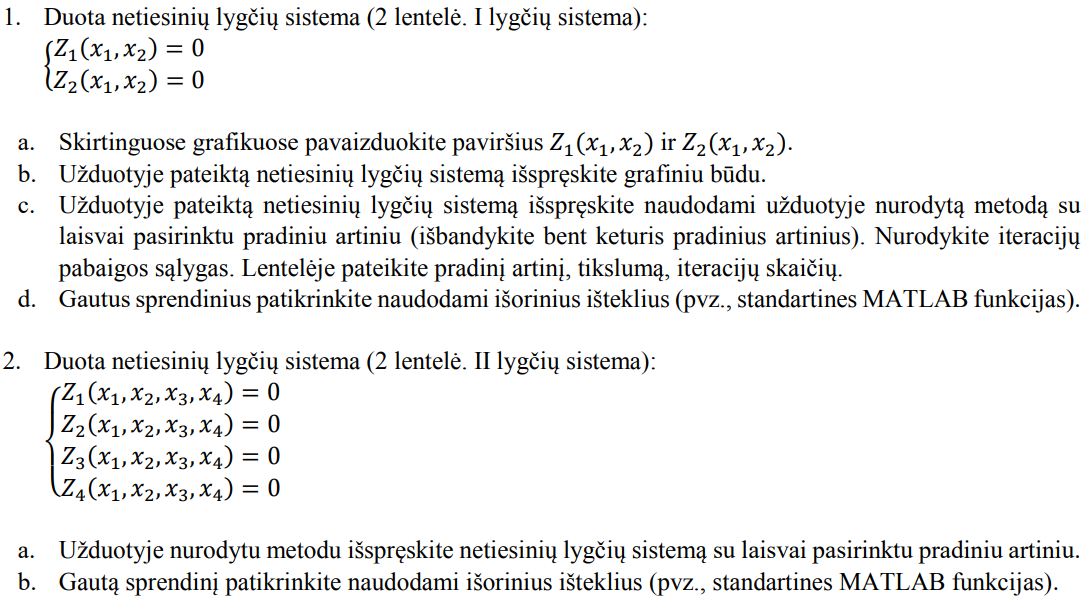
# ]

A = np.matrix(A\_nedalinta).astype(np.float)

B = np.matrix(B\_nedalinta).transpose().astype(np.float)

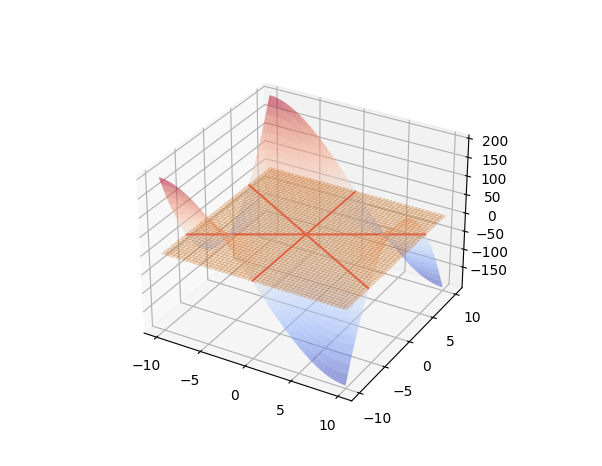
task1(A, B)

## Antra užduotis. Netiesinių lygčių sistemų sprendimas. Pirma dalis

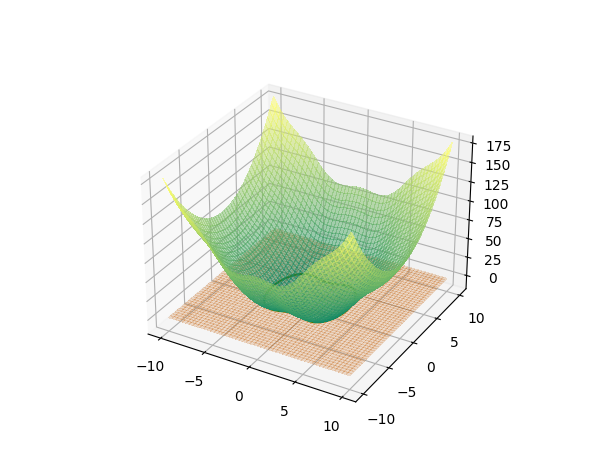


pav. 5 antroji užduotis

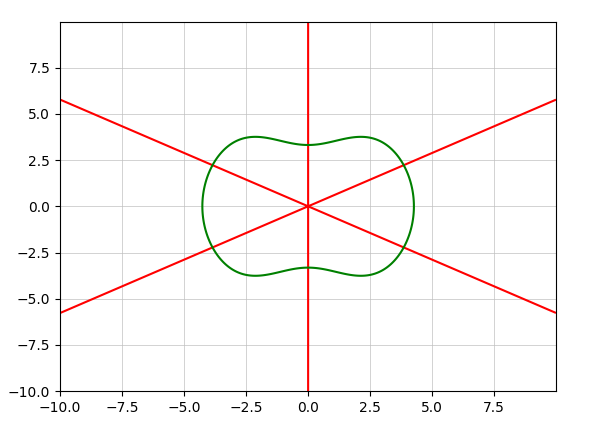
### Grafinis sprendimas



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, alpha=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, alpha=0.2)

cp = aXis.contour(X, Y, Z2, levels=0, colors='green')

plt.show()

figure = plt.figure()

aXis = figure.gca()

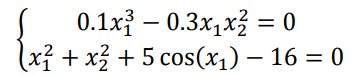
aXis.grid(color='#C0C0C0', linestyle='-', linewidth=0.5)

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)



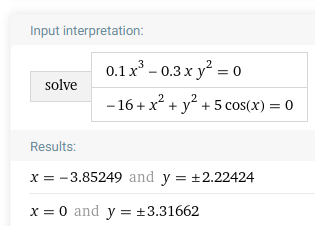
Parametrai:

|  |  |  |
| --- | --- | --- |
| iterationMax | step | eps |
| 50 | 0.01 | 1e-8 |

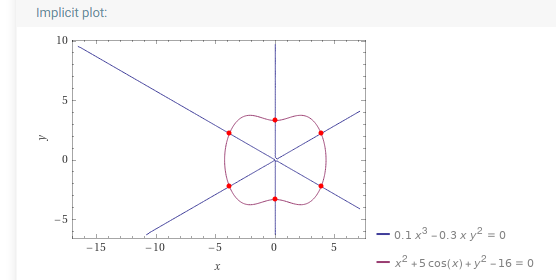
Rezultatai:

|  |  |  |  |
| --- | --- | --- | --- |
| 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 esama, 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"

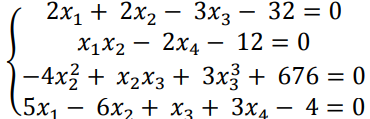
# print(descent(-4, 2, 50, 0.01, 1e-8))

# 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))

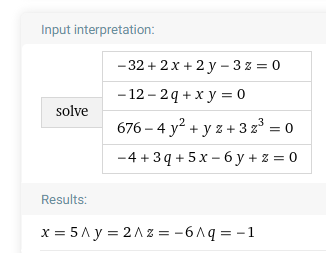
### Keturių lygčių sistema (antra lentelė)



|  |  |  |
| --- | --- | --- |
| 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,  -5.999847520560582,  -0.9937586510871118) | 9.99838e-06 |

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, x4, precision))

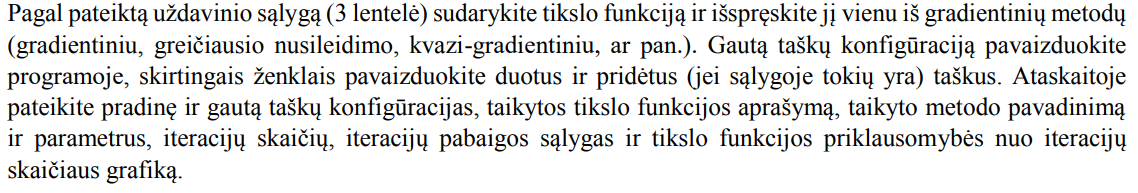
        if (precision < eps):

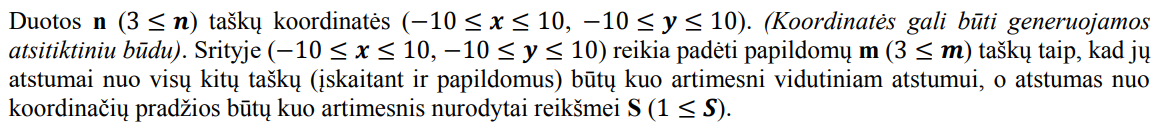
            return x1, x2, x3, x4

    return "tikslumas nepasiektas"

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

## Trečia užduotis. Optimizavimas





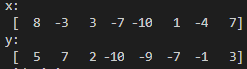
Tikslo funkcija:

Taikytas metodas: minimizavimas priešinga gradientui kryptimi.

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

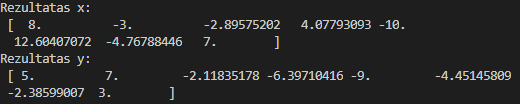
Pabaigos sąlyga: ieškoma kol tenkinamas pasirinktas tikslumas arba jeigu pasiektas iteracijų limitas, nutraukiamas metodas.

Pradiniai duomenys:

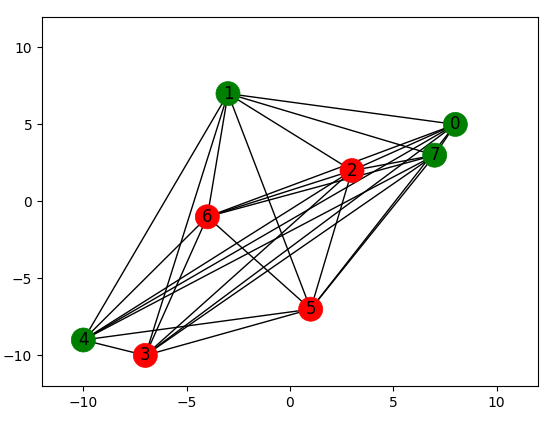


pav. 12 Pradiniai duomenys

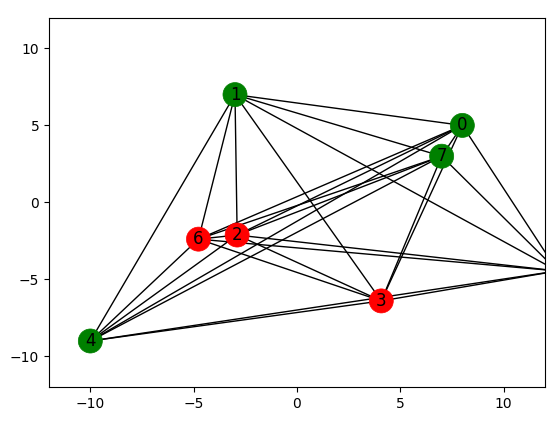
Rezultatai:



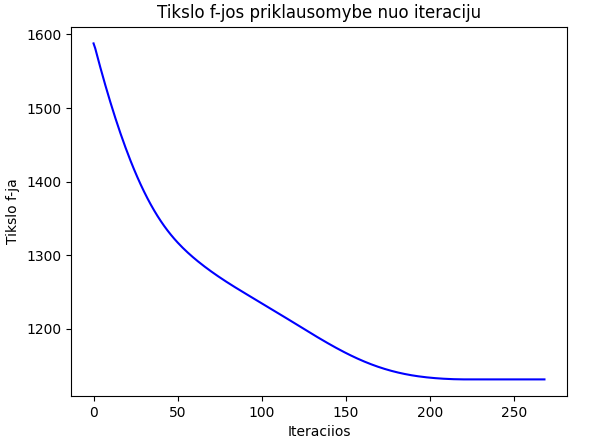
pav. 13 Rezultatai



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.abs(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()