Kauno technologijos universitetas

Informatikos fakultetas

P170B115 Skaitiniai metodai ir algoritmai

2 projektinė užduotis. Lygčių sistemų sprendimas ir optimizavimas.

Nedas Liaudanskis

Studentas

doc. Čalnerytė Dalia

Dėstytoja

KAUNAS, 2023

Turinys

Įvadas

Projekto užduotis, suskirstyta į tris dalis, susijusi su trimis pagrindiniais tikslais: rasti sprendinius tiesinėms ir netiesinėms sistemoms, taip pat spręsti optimizavimo problemą.

Pirmoje dalyje nagrinėsime tiesinės lygčių sistemas ir ieškosime jų sprendinių naudodami skaitinius metodus, tokius kaip Gauso eliminacijos metodas, Gauso-Žordano iteracija, Gauso-Zeidelio iteracija ir paprastųjų iteracijų metodas. Be to, išmoksime taikyti tiesinės lygčių sistemos skaidą naudojant LU, QR ir Choleskio dekompozicijas. Tirsimos situacijos, kai lygčių sistema turi begalę sprendinių arba neturi sprendinių.

Antrajame etape dėmesys bus skirtas netiesinėms lygčių sistemoms, ir mes bandysime rasti jų sprendinius naudodami skirtingus skaitinius metodus, tokius kaip greičiausio nusileidimo metodas, paprastųjų iteracijų metodas, Niutono metodas ir modifikuotas Niutono metodas. Taip pat svarbu išmokti tinkamai pasirinkti pradinius artinius ir apibrėžti sąlygas, kurios nurodys, kada galime baigti skaičiavimus.

Paskutinėje dalyje spręsime optimizavimo problemą, naudodami pasirinktą gradientinį optimizavimo metodą. Be to, patikrinsime gautus sprendinius naudodami išorinių išteklių funkcijas.

1. Dalis, Tiesinių lygčių sistemų sprendimas

A) Dalis. Reikia iš pirmoje lentelėje esančių tiesinių lygčių sistemų paskirstytų pagal savo variantą suprogramuoti .py kodą, kuris galės išspręsti šias tiesines lygčių sistemas ir išvesti jų sprendinius į ekraną. Gautus sprendinius būtina patikrinti naudojant kitus metodus.

Variantas: 10

Užduoties Nr.	Metodas	Lygčių sistemų Nr. (1 lentelė)
10	Atspindžio	8, 13, 20
	Paprastųjų iteracijų	8

Duotos lygčių sistemos:

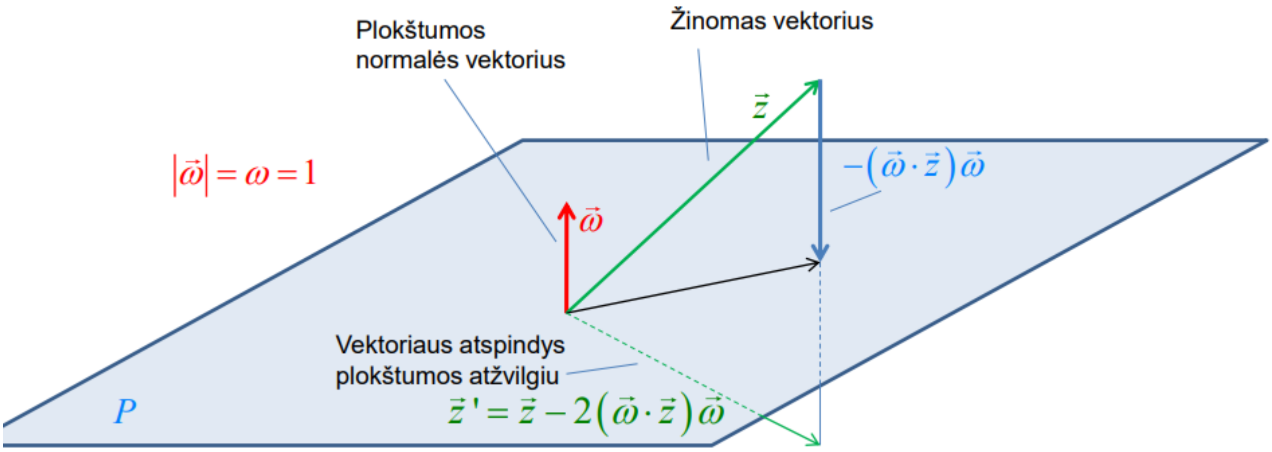
8	$\begin{cases} 4x_1 + 3x_2 - x_3 + x_4 = 12 \\ 3x_1 + 9x_2 - 2x_3 - 2x_4 = 10 \\ -x_1 - 2x_2 + 11x_3 - x_4 = -28 \\ x_1 - 2x_2 - x_3 + 5x_4 = 16 \end{cases}$

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

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

Atspindžio metodas:

Tai atspindžio metodas yra pranašesnis už Gauso metodą, es yra sugeneruojamos mažesnės paklaidos.



Atspindžio algoritmo idėja:

Pagal formulę $\{\boldsymbol{z}'\} = [\boldsymbol{Q}]\{\boldsymbol{z}\}$, $[\boldsymbol{Q}] = [\boldsymbol{E}] - \mathbf{2}\{\omega\}\{\omega^{t}\}$
Vektorius {z} pertvarkomas į jo atspindį atžvilgiu plokštumos, kurios normalės vektorius yra {ω}

Pagal formulę $\{\omega\} = \frac{\{z\}-\{z'\}}{\|\{z\}-\{z'\}\|}$ nustatoma normalė veidrodžio plokštumos, kuris atspindi duotą vektorių į duotą jo atspindį. Duotųjų atspindimo {z} vektoriaus ir atspindžio vektorių {z'} Euklido normos turi būti lygios, t.y $\|\{\boldsymbol{z}\}\| = [\![\{\boldsymbol{z}'\}]\!]$

Kodas:

```
import numpy as np

# Apibrėžiame koeficientų matricą A ir dešinės pusės vektorių b
#8 Var
#A = np.array([[4, 3, -1, 1],
#              [3, 9, -2, -2],
#              [-1, -2, 11, -1],
#              [1, -2, -1, 5]], dtype=float)
#b = np.array([12, 10, -28, 16], dtype=float).reshape(-1, 1)
#13 Var
A = np.array([[1, -2, 3, 4],
              [1, 0, -1, 1],
              [2, -2, 2, 5],
```

```
            [0, -7, 3, 1]], dtype=float)

b = np.array([11, -4, 7, 2], dtype=float).reshape(-1, 1)

#20 Var
#A = np.array([[2, 4, 6, -2],
#             [1, 3, 1, -3],
#             [1, 1, 5, 1],
#             [2, 3, -3, -2]], dtype=float)
#
#b = np.array([2, 1, 7, 2], dtype=float).reshape(-1, 1)


n = A.shape[0]  # Lygčių skaičius

# Tikriname, ar matrica yra singuliari, remdamiesi jos determinanto artimumu nuliui
det_A = np.linalg.det(A)
if abs(det_A) < 1e-10:
    print("Matrica A yra singuliari. Jos neturi unikalio sprendinio.")
else:
    # Papildome matricą A stulpeliu b
    A = np.hstack((A, b))

    # Perėjimas į priekį
    for i in range(n):
        # Dalinis perstūmimas (partial pivoting)
        max_row = np.argmax(np.abs(A[i:n, i])) + i
        A[[i, max_row]] = A[[max_row, i]]

        # Tikriname nulinį perstūmą (singuliarumą)
        if abs(A[i, i]) < 1e-10:
            print("Matrica yra singuliari. Jos neturi unikalio sprendinio.")
            break

        for j in range(i + 1, n):
            faktorius = A[j, i] / A[i, i]
            A[j, i:] = A[j, i:] - faktorius * A[i, i:]

    if abs(A[-1, -2]) < 1e-10:
        print("Matrica yra singuliari. Jos neturi unikalio sprendinio.")
    else:
        # Atgalinė substitucija
        x = np.zeros(n)
        for i in range(n - 1, -1, -1):
            x[i] = (A[i, -1] - np.sum(A[i, i+1:n] * x[i+1:n])) / A[i, i]
```

```
print("Sprendinys:")
print(x)
```

Rezultatai:

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

```
Sprendinys:
[ 1.  1. -2.  3.]
```

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

```
Matrica A yra singuliari. Jos neturi unikalio sprendinio.
```

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

```
Matrica A yra singuliari. Jos neturi unikalio sprendinio.
```

Paprastųjų iteracijų metodas:

Iteracijų metodo idėja pasiiriame tuo, jog mes galime pasirinkti pradinį artinį, kurį naudojant tam tikrose formulėse, galėsime artėti prie reikiamų sistemų sprendimų. Visas skaičiavimas pagrįstas viena formule:

$$\{\mathbf{x}\}^{(k+1)} = [\boldsymbol{\alpha}]^{-1}\left(\{\tilde{\mathbf{b}}\} - [\tilde{\mathbf{A}}]\ \{\mathbf{x}\}^{(k)}\right)$$

Tai yra paprastųjų iteracijų algoritmas, kurio dėka galime apskaičiuoti sekantį artinį.

k – raidė reprezentuoja iteracijų skaičių, kuris mums padeda nustatyti tikslumą, kurį norime gauti naudojant šį iteracijų algoritmą.

Tikslumo formulė(iteracijos pabaigos sąlygą):

$$\frac{\left\| \left\{ \mathbf{x} \right\}^{(k+1)} - \left\{ \mathbf{x} \right\}^{(k)} \right\|}{\left\| \left\{ \mathbf{x} \right\}^{(k+1)} \right\| + \left\| \left\{ \mathbf{x} \right\}^{(k)} \right\|} < \varepsilon$$

Iteracijas vykdome tol, kol ši sąlyga nėra išpildyta.

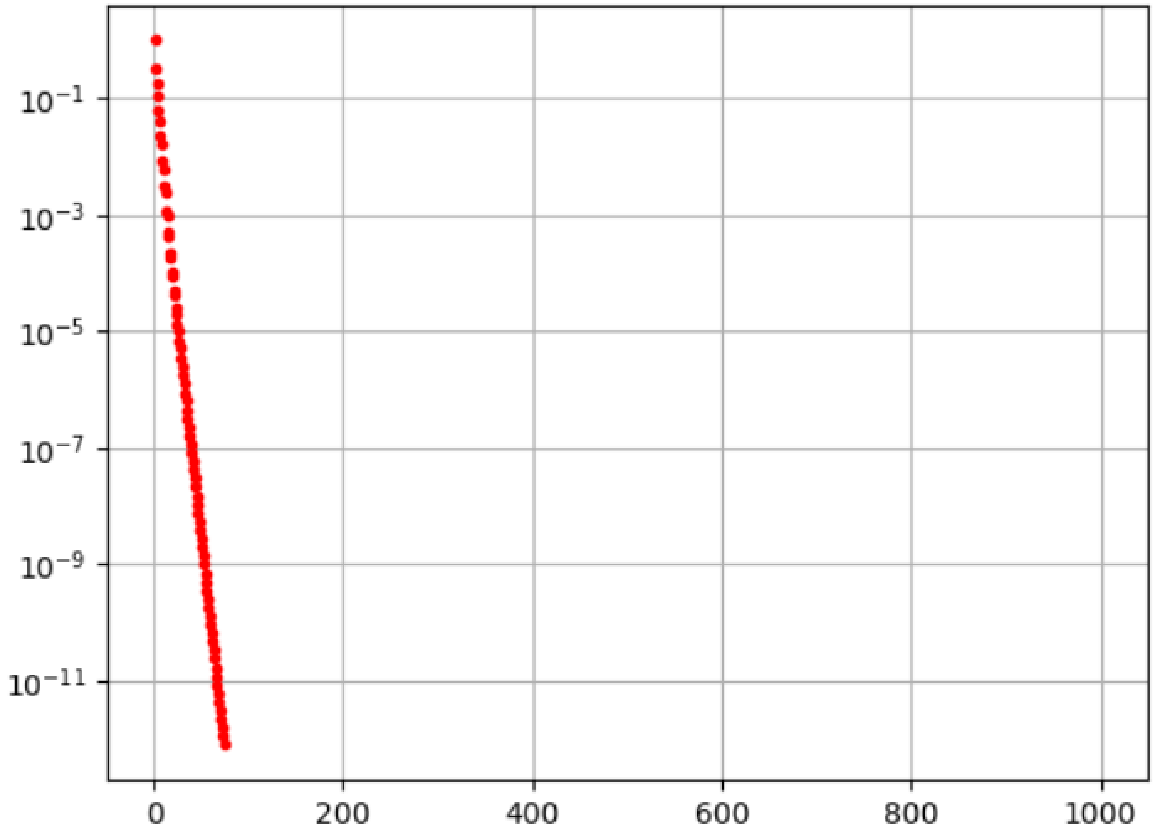
Kodas:

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

A = np.array([[4, 3, -1, 1],
              [3, 9, -2, -2],
              [-1, -2, 11, -1],
              [1, -2, -1, 5],], dtype=float)

b = np.array([12, 10, -28, 16], dtype=float)

n = A.shape[0]

method = 'simple_iterations'
alpha = np.array([1, 1, 1, 1], dtype=float)  # Use your desired values for alpha

Atld = (np.diag(1.0 / np.diag(A)).dot(A) - np.diag(alpha))
btld = np.diag(1.0 / np.diag(A)).dot(b)

nitmax = 1000
eps = 1e-12
x = np.zeros(n, dtype=float)
x1 = np.zeros(n, dtype=float)
prec = np.zeros(nitmax, dtype=float)

print('Solving using simple iterations:')
for it in range(nitmax):
    x1 = (btld - Atld.dot(x)) / alpha
    prec[it] = np.linalg.norm(x1 - x) / (np.linalg.norm(x) + np.linalg.norm(x1))

    if prec[it] < eps:
        print('Alpha:', alpha)
        print('Solution:', x)
        print('Check:', btld.dot(x) - b)
        break

    x = x1
else:
    print('Method did not converge')

plt.semilogy(range(1, len(prec) + 1), prec, 'r.')
plt.grid(True)
plt.show()
```

Rezultatai:

```
Solving using simple iterations:
```

Alpha: [1. 1. 1. 1.]
Solution: [1. 1. -2. 3.]
Check: [1.61435310e-11 -2.73256973e-11 -7.11253278e-12 -1.51096913e-11]



B) dalis. Reikia trečioje lentelėje duotą tiesinę lygčių sistemą išspręsti su duotu sprendimo metodu arba LU arba QR

Variantas: 10

Užduoties Nr.	Lygčių sistema	b1	B2	B3	Metodas
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

LU metodo paaiškinimas:

Šis metodas skaido mums gerai žino tiesinių sistemų formulę $\boldsymbol{AX} = \boldsymbol{B}$

Pirmiausia A = LU, viršutinė ir apatinė matricos, jos sudarys visa pilną matricą.

Tada įsistatę gauname LUX=B.

Gavus tokią formulę skaidome į dvi dalis:

UX = Y;

LY = B ir UX = Y

Kai gauname šias formules, mums reikia pirmiausia išspręsti LY=B, tada gavus Y galime rasti ir X iš formulės UX=Y.

Kodas:

```python
import numpy as np

A = np.array([[6, 1, 3, -2],
              [6, 8, 1, -1],
              [12, -2, 4, -1],
              [8, 1, 1, 5],], dtype=float)

#b = np.array([8, 14, 13, 15], dtype=float)
#b = np.array([67, 77, 126, 95], dtype=float)
b = np.array([-5.25, 0, -13.5, 7.25], dtype=float)
n = A.shape[0]

L = np.eye(n)
U = np.zeros((n, n), dtype=float)

U[0, :] = A[0, :]

for i in range(n - 1):
    for j in range(i + 1, n):
        r = A[j, i] / A[i, i]

        U[j, i:] = A[j, i:] - A[i, i:] * r
        L[j, i] = r

        A[j, i+1:] = A[j, i+1:] - A[i, i+1:] * r
        A[j, i] = r

print("Matrix A:")
print(A)
print("Matrix L:")
print(L)
print("Matrix U:")
print(U)

# Check if A = LU
if np.allclose(A, np.dot(L, U)):
    print("LU factorization is correct.")
else:
    print("LU factorization is incorrect.")

y = np.zeros(n, dtype=float)
for i in range(n):
    y[i] = b[i] - np.dot(L[i, :i], y[:i])

# Backward Substitution
x = np.zeros(n, dtype=float)
for i in range(n - 1, -1, -1):
    x[i] = (y[i] - np.dot(U[i, i + 1:], x[i + 1:])) / U[i, i]

print("Solution x:")
print(x)
```

Rezultatai:

```
Kai B = [8, 14, 13, 15]
Matrix A:
[[ 6.          1.          3.         -2.        ]
 [ 1.          7.         -2.          1.        ]
 [ 2.         -0.57142857 -3.14285714  3.57142857]
 [ 1.33333333 -0.04761905  0.98484848  4.1969697 ]]
Matrix L:
[[ 1.          0.          0.          0.        ]
 [ 1.          1.          0.          0.        ]
 [ 2.         -0.57142857  1.          0.        ]
```

```
 [ 1.33333333 -0.04761905  0.98484848  1.        ]]
Matrix U:
[[ 6.          1.          3.         -2.        ]
 [ 0.          7.         -2.          1.        ]
 [ 0.          0.         -3.14285714  3.57142857]
 [ 0.          0.          0.          4.1969697 ]]
LU factorization is incorrect.
Solution x:
[1. 1. 1. 1.]
```

Kai B = [67, 77, 126, 95]

```
Matrix A:
[[ 6.          1.          3.         -2.        ]
 [ 1.          7.         -2.          1.        ]
 [ 2.         -0.57142857 -3.14285714  3.57142857]
 [ 1.33333333 -0.04761905  0.98484848  4.1969697 ]]
Matrix L:
[[ 1.          0.          0.          0.        ]
 [ 1.          1.          0.          0.        ]
 [ 2.         -0.57142857  1.          0.        ]
 [ 1.33333333 -0.04761905  0.98484848  1.        ]]
Matrix U:
[[ 6.          1.          3.         -2.        ]
 [ 0.          7.         -2.          1.        ]
 [ 0.          0.         -3.14285714  3.57142857]
 [ 0.          0.          0.          4.1969697 ]]
LU factorization is incorrect.
Solution x:
[10.  2.  3.  2.]
```

Kai B = [-5.25, 0, -13.5, 7.25]

```
Matrix A:
[[ 6.          1.          3.         -2.        ]
 [ 1.          7.         -2.          1.        ]
 [ 2.         -0.57142857 -3.14285714  3.57142857]
 [ 1.33333333 -0.04761905  0.98484848  4.1969697 ]]
Matrix L:
[[ 1.          0.          0.          0.        ]
 [ 1.          1.          0.          0.        ]
 [ 2.         -0.57142857  1.          0.        ]
 [ 1.33333333 -0.04761905  0.98484848  1.        ]]
Matrix U:
[[ 6.          1.          3.         -2.        ]
 [ 0.          7.         -2.          1.        ]
 [ 0.          0.         -3.14285714  3.57142857]
 [ 0.          0.          0.          4.1969697 ]]
LU factorization is incorrect.
Solution x:
[-1.91606498  1.37815884  3.92599278  3.45487365]
```

2. Dalis. Netiesinių lygčių sistemų sprendimas

Duota netiesinių lygčių sistema:

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

1. Reikia skirtinguose grafikuose pavaizduoti abu paviršius.

2. Pateiktą netiesinių lygčių sistemą išspręsti grafiniu būdu.
3. Grafiškai atvaizduoti susikirtimo taškus
4. Sudaryti pradinių artinių tinklelį naudojant grafines priemones.

Nr.	Lygčių sistema	Metodas
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

Uždavinį reikės spręsti naudojant Broideno metodą. Šio metodo idėja yra tokia, Jog, kas keletą žingsnių, artinyje reikia skaičiuoti Jakobio matrica, kad žinotumėme kaip reikia keisti argumentus, norint gauti sekantį artinį. Taip pat reikia žinoti, kiekvienai funkcijai priklausančią Teiloro eilutę. Teiloro eilutės formulė vienai funkcijai:

$$f_k(\mathbf{x}+\Delta\mathbf{x}) \approx f_k(\mathbf{x}) + \Delta x_1 \left.\frac{\partial f_k}{\partial x_1}\right|_{\mathbf{x}} + \Delta x_2 \left.\frac{\partial f_k}{\partial x_2}\right|_{\mathbf{x}} + \ldots + \Delta x_n \left.\frac{\partial f_k}{\partial x_n}\right|_{\mathbf{x}},$$

Jakobo matrica:

$$\begin{Bmatrix} f_1(\mathbf{x}+\Delta\mathbf{x}) \\ f_2(\mathbf{x}+\Delta\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}+\Delta\mathbf{x}) \end{Bmatrix} = \begin{Bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{Bmatrix} + \left.\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}\right|_{\mathbf{x}} \begin{Bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_n \end{Bmatrix}$$

Norint gauti sekantį prieaugį, reikia naudoti šią formulę:

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \Delta\mathbf{x}$$

Joje mums reikia gauti delta X, tam bus naudojama Jakobio matrica ir padarius supaprastinimus gauname galutinę formulę:

$$\mathbf{x}^{i+1} = \mathbf{x}^{i} - \left[\left. \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right|_{\mathbf{x}^{i}} \right]^{-1} \mathbf{f}(\mathbf{x}^{i})$$

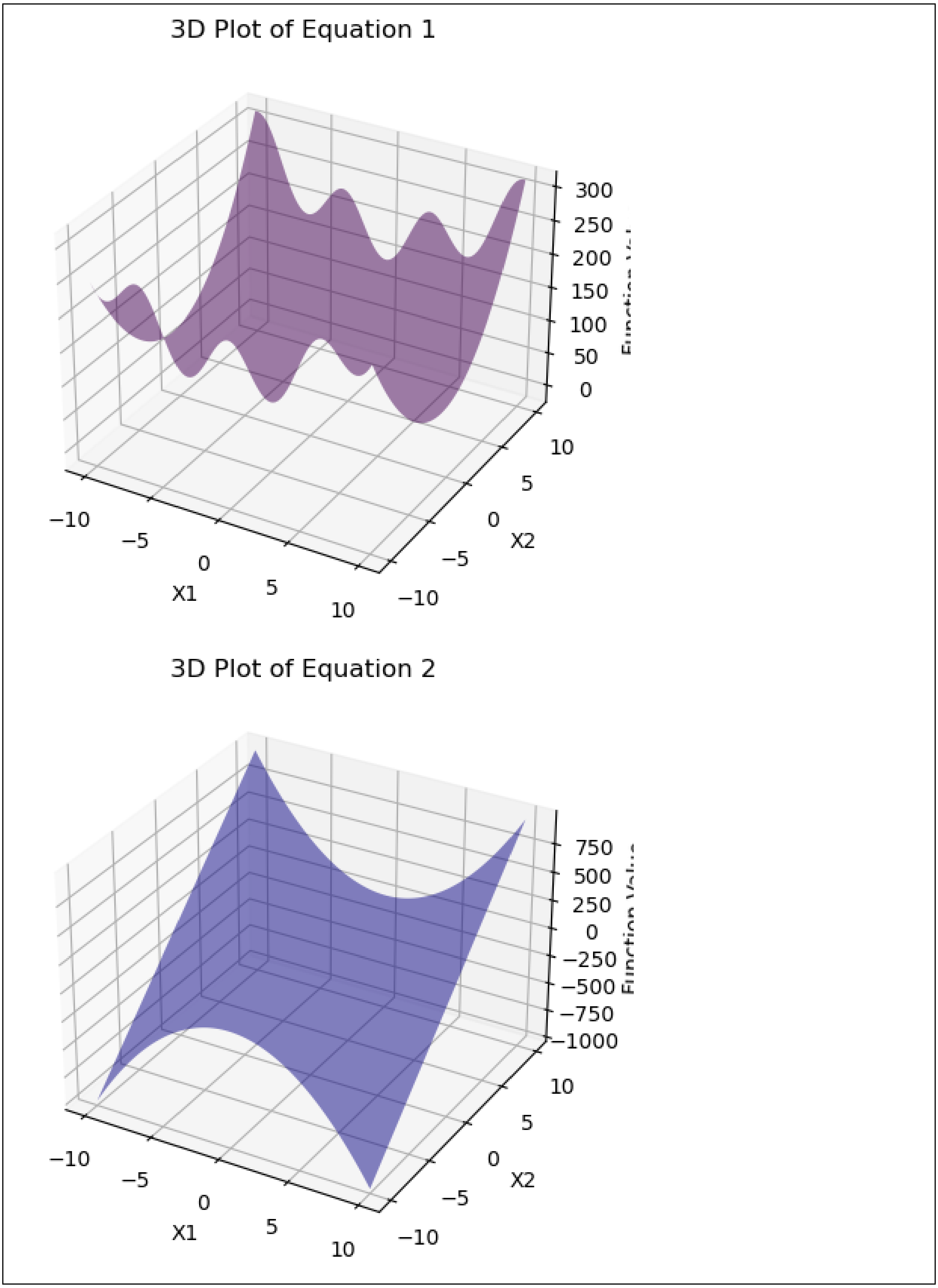
Broideno metodas nuo Niutono skiriasi tik tuo, jog Jakobio matricą, mes atnaujiname tik kas keletą žingsnių.

1) <u>Dalis, paviršių atvaizdavimas</u>

<u>Kodas</u>**:**

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the range of x1 and x2 values
x1 = np.linspace(-10, 10, 100)
x2 = np.linspace(-10, 10, 100)

# Create a grid of (x1, x2) values
X1, X2 = np.meshgrid(x1, x2)

# Define the equations
equation1 = X1**2 + 2 * (X2 - np.cos(X1))**2 - 20
equation2 = X1**2 * X2 - 2

# Create a 3D plot for equation 1
fig1 = plt.figure()
ax1 = fig1.add_subplot(111, projection='3d')
ax1.plot_surface(X1, X2, equation1, alpha=0.5, rstride=100, cstride=100, cmap='viridis', edgecolor='none')
ax1.set_xlabel('X1')
ax1.set_ylabel('X2')
ax1.set_zlabel('Function Value')
ax1.set_title('3D Plot of Equation 1')
ax1.grid(True)

# Create a 3D plot for equation 2
fig2 = plt.figure()
ax2 = fig2.add_subplot(111, projection='3d')
ax2.plot_surface(X1, X2, equation2, alpha=0.5, rstride=100, cstride=100, cmap='plasma', edgecolor='none')
ax2.set_xlabel('X1')
ax2.set_ylabel('X2')
ax2.set_zlabel('Function Value')
ax2.set_title('3D Plot of Equation 2')
ax2.grid(True)

plt.show()
```

<u>Rezultatai:</u>



2) Dalis, netiesinių lygčių sprendimas grafiniu būdu

Kodas**:**

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

# Define the precision function
def precision(x1, x2, f1, f2, eps):
    s = np.abs(x1 - x2) / (np.abs(x1) + np.abs(x2) + np.abs(f1) + np.abs(f2))
    return s

# Surface function
def Surface(X, Y, LFF):
    siz = np.shape(X)
    Z = np.zeros(shape=(siz[0], siz[1], 2))
    for i in range(siz[0]):
        for j in range(siz[1]):
            Z[i, j, :] = LFF([X[i][j], Y[i][j]]).T
    return Z

# System of equations function
def SystemOfEquations(x):
    s = np.array([
        x[0]**2 + 2*(x[1] - np.cos(x[0]))**2 - 20,
        x[0]**2 * x[1] - 2
    ])
    s = s.reshape((2, 1))
    s = np.matrix(s)
    return s

n = 2  # Number of equations
x = np.zeros((n, 1))
x[0] = 2.5
x[1] = 0.3

maxiter = 30  # Maximum allowed iterations
eps = 1e-6  # Required precision

# Graph: LF surfaces
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_zlabel('z')
ax2 = fig1.add_subplot(1, 2, 2, projection='3d')
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.set_zlabel('z')
```

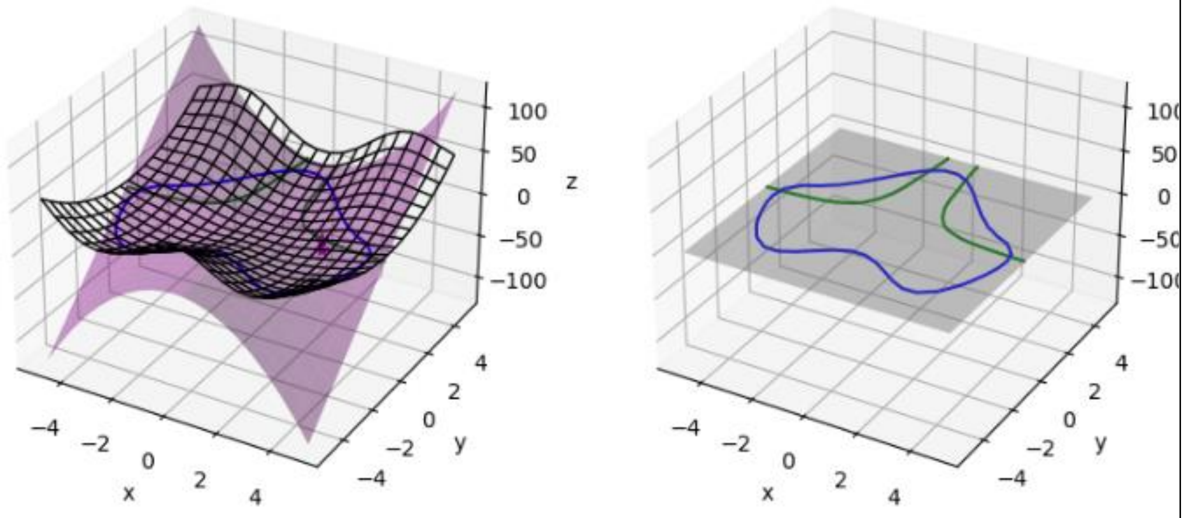
```
plt.draw()
xx = np.linspace(-5, 5, 20)
yy = np.linspace(-5, 5, 20)
X, Y = np.meshgrid(xx, yy)
Z = Surface(X, Y, SystemOfEquations)

wire1 = ax1.plot_wireframe(X, Y, Z[:, :, 0], color='black', alpha=1, linewidth=1, antialiased=True)
surf2 = ax1.plot_surface(X, Y, Z[:, :, 1], color='purple', alpha=0.4, linewidth=0.1, 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)

dx = 0.1  # Initial step for Jacobian matrix
A = np.zeros((n, n))
x1 = np.zeros((n, 1))

for i in range(n):
    x1 = x.copy()
    x1[i] += dx
    A[:, i] = (SystemOfEquations(x1) - SystemOfEquations(x)).flatten() / dx

ff = SystemOfEquations(x)
ax1.plot3D([x[0, 0], x[0, 0], x[0, 0]], [x[1, 0], x[1, 0], x[1, 0]], [0, ff[0, 0], 0], "m*-")
plt.draw()
plt.pause(1)

print("\nInitial Jacobian matrix:")
print(A)
print(ff.T)

for i in range(1, maxiter):
    deltax = -np.linalg.solve(A, ff)
    x1 = x + deltax
    ff1 = SystemOfEquations(x1)
```

```
    A += (ff1 - ff - A @ deltax) @ deltax.T / (deltax.T @ deltax)
    precision_value = precision(x, x1, ff, ff1, eps)
    print(precision_value)
    ff = ff1
    x = x1

    if (precision_value < eps).all():
        break

    ax1.plot3D([x[0, 0], x1[0, 0], x[0, 0]], [x[1, 0], x1[1, 0], x[1, 0]], [0, 0, 0], "ro-")
    ax1.plot3D([x[0, 0], x1[0, 0], x[0, 0]], [x[1, 0], x1[1, 0], x[1, 0]], [ff[0, 0], ff1[0, 0], 0], "c-
.")
    ax1.plot3D([x1[0, 0], x1[0, 0], x1[0, 0]], [x1[1, 0], x1[1, 0], x1[1, 0]], [0, 0, ff1[0, 0]], "m*-
")
    ax2.plot3D([x[0, 0], x1[0, 0], x[0, 0]], [x[1, 0], x1[1, 0], x[1, 0]], [0, 0, 0], "ro-")
    plt.draw()
    plt.pause(2)

ax1.plot3D([x[0, 0], x[0, 0]], [x[1, 0], x[1, 0]], [0, 0], "ks")
ax2.plot3D([x[0, 0], x[0, 0]], [x[1, 0], x[1, 0]], [0, 0], "ks")
plt.draw()
plt.pause(1)

print("Solution:")
print(x1.T)
print("Final precision:")
print(precision_value)

print("Plotting pre-finished")
plt.show()
print("Plotting finished")
```

Rezultatai:

```
Initial Jacobian matrix:
[[7.61748651 4.60457446]
 [1.53       6.25      ]]
[[-11.32496548  -0.125      ]]
[[0.08697922]
 [0.09207081]]
<Figure size 640x480 with 0 Axes>
[[0.01171792]
 [0.05554021]]
<Figure size 640x480 with 0 Axes>
[[0.02836279]
 [0.04077012]]
<Figure size 640x480 with 0 Axes>
[[0.00963202]
 [0.04560642]]
<Figure size 640x480 with 0 Axes>
```

```
[[0.00299555]
 [0.01236304]]
<Figure size 640x480 with 0 Axes>
[[0.0002219 ]
 [0.01402251]]
<Figure size 640x480 with 0 Axes>
[[7.47293949e-05]
 [3.94003267e-03]]
<Figure size 640x480 with 0 Axes>
[[3.17499143e-06]
 [1.53685188e-04]]
<Figure size 640x480 with 0 Axes>
[[5.10328488e-08]
 [2.70579381e-06]]
<Figure size 640x480 with 0 Axes>
[[4.71839964e-10]
 [2.39582124e-08]]
<Figure size 640x480 with 0 Axes>
Solution:
[[4.44160772 0.10137937]]
Final precision:
[[4.71839964e-10]
 [2.39582124e-08]]
Plotting pre-finished
Plotting finished
```

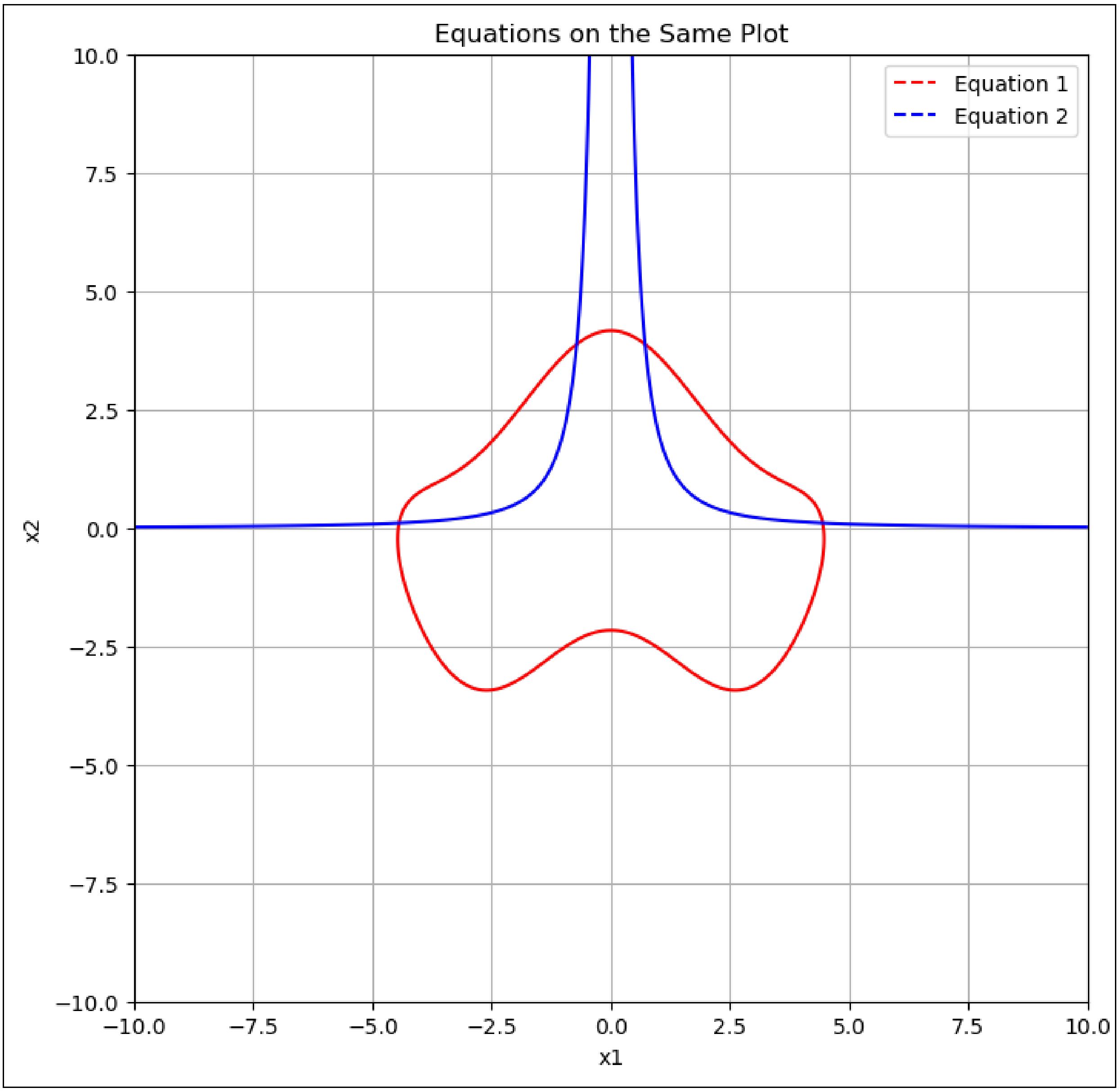


3) Dalis, susikirtimo taškų diagrama.

Kodas:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

# Define the range for x1 and x2
x1 = np.linspace(-10, 10, 400)
x2 = np.linspace(-10, 10, 400)

# Create a grid of values for x1 and x2
X1, X2 = np.meshgrid(x1, x2)

# Define the equations
equation1 = X1**2 + 2 * (X2 - np.cos(X1))**2 - 20
equation2 = X1**2 * X2 - 2

# Create a contour plot for each equation
plt.figure(figsize=(8, 8))
contour1 = plt.contour(X1, X2, equation1, levels=[0], colors='r')
contour2 = plt.contour(X1, X2, equation2, levels=[0], colors='b')

# Create custom legend labels
legend_elements = [
    Line2D([0], [0], color='r', label='Equation 1', linestyle='--'),
    Line2D([0], [0], color='b', label='Equation 2', linestyle='--')
]

plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Equations on the Same Plot')
plt.legend(handles=legend_elements, loc='upper right')
plt.grid(True)

plt.show()
```

Rezultatai:

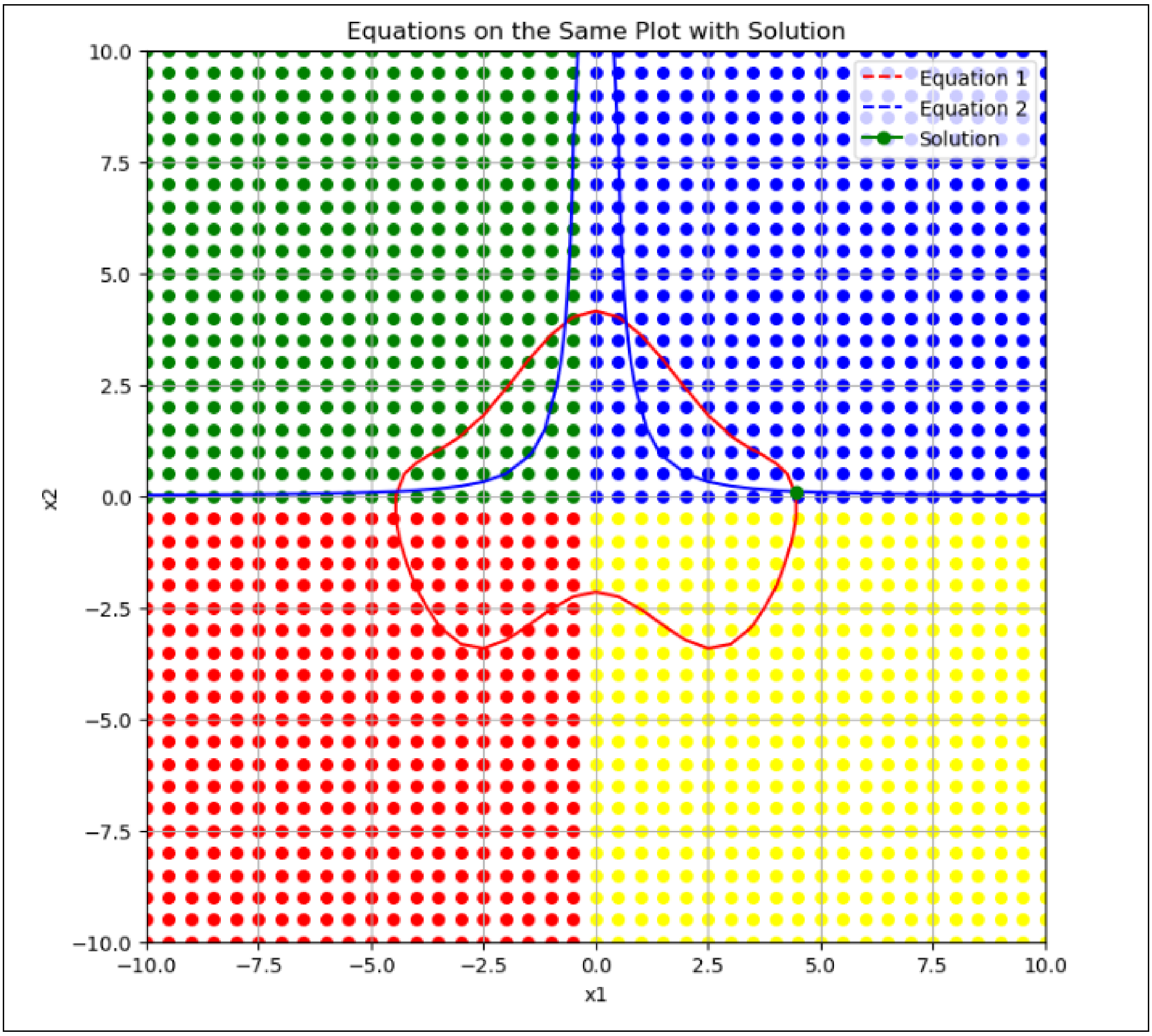


4) Dalis, pradinių artinių tinklelis

Kodas:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

#Solution
Solution = np.array([[4.44160772, 0.10137937]])

#Define the range for x1 and x2
x1 = np.arange(-10, 10.5, 0.5)
x2 = np.arange(-10, 10.5, 0.5)

#Create a grid of values for x1 and x2
X1, X2 = np.meshgrid(x1, x2)

#Define the equations
equation1 = X1**2 + 2 * (X2 - np.cos(x1))**2 - 20
equation2 = X1**2 * X2 - 2

#Create a contour plot for each equation
plt.figure(figsize=(8, 8))
contour1 = plt.contour(X1, X2, equation1, levels=[0], colors='r')
contour2 = plt.contour(X1, X2, equation2, levels=[0], colors='b')

#Plot the solution point
plt.plot(Solution[0, 0], Solution[0, 1], 'go', label='Solution')

#Define and plot quadrant dots with different colors
for i in range(len(x1)):
    for j in range(len(x2)):
        if x1[i] < 0 and x2[j] < 0:
            plt.scatter(x1[i], x2[j], c='red', s=30)
        elif x1[i] < 0 and x2[j] >= 0:
            plt.scatter(x1[i], x2[j], c='green', s=30)
        elif x1[i] >= 0 and x2[j] < 0:
            plt.scatter(x1[i], x2[j], c='yellow', s=30)
        else:
            plt.scatter(x1[i], x2[j], c='blue', s=30)

#Create custom legend labels
legend_elements = [
    Line2D([0], [0], color='r', label='Equation 1', linestyle='--'),
    Line2D([0], [0], color='b', label='Equation 2', linestyle='--'),
    Line2D([0], [0], color='g', marker='o', label='Solution')
]

plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Equations on the Same Plot with Solution')
plt.legend(handles=legend_elements, loc='upper right')
plt.grid(True)

plt.show()
```

<u>Rezultatai:</u>



3. Dalis. Optimizavimas

Pagal duotą uždavinio sąlygą reikia aprašyti uždavinį ir jį išspręsti vienu iš gradiento metodų (gradientiniu, greičiausio nusileidimo). Gautą taškų konfigūraciją atvaizduoti programoje.

1. **Pradine konfigūracija:** Pradinės konfigūracijos taškai yra sugeneruojami atsitiktinai. Jau turime 10 prekybos vietų (nustatyta kaip n = 10), ir mes stengiamės pridėti dar 2 naujas prekybos vietas (nustatyta kaip optNodes = 2). Pradinės vietos ir naujos vietos yra atsitiktinai sugeneruojamos lauke nuo -10 iki 10.
2. **Optimizavimas:** Optimizavimas vyksta naudojant gradientinio nusileidimo metodą.
3. **Iteracijos pabaigos sąlyga:** Pradžioje nustatoma pradinė žingsnio ilgis (step) ir tikslumas (eps). Tada pradedamas iteracijų procesas. Tikslas yra minimizuoti tikslo funkciją keičiant naujų prekybos vietų pozicijas. Po kiekvienos iteracijos apskaičiuojamas gradiento vektorius, ir prekybos vietos pozicijos yra atnaujinamos pagal gradientinio nusileidimo formulę. Jei nauja tikslo funkcijos vertė didesnė nei ankstesnė, žingsnis sumažinamas. Procesas tęsiamas tol, kol pasiekiama nustatyta žingsnio ilgio riba, gradiento norma tampa mažesnė nei nustatytas tikslumas (eps), arba pasiekiama maksimalus iteracijų skaičius (1000).

Kodas:

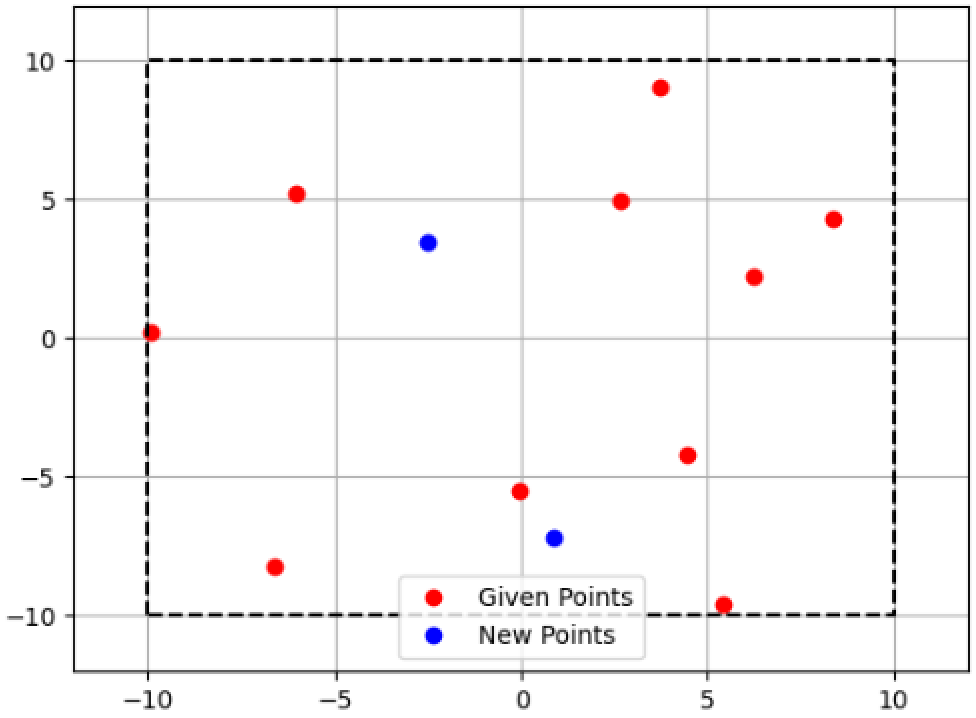
```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(10)

# Define the precision function
def precision(x1, x2, f1, f2, eps):
    s = np.abs(x1 - x2) / (np.abs(x1) + np.abs(x2) + np.abs(f1) + np.abs(f2))
    return s

# Surface function
def Surface(X, Y, LFF):
    siz = np.shape(X)
    Z = np.zeros(shape=(siz[0], siz[1], 2))
    for i in range(siz[0]):
        for j in range(siz[1]):
            Z[i, j, :] = LFF([X[i][j], Y[i][j]]).T
    return Z

# System of equations function for the cost
def SystemOfEquationsCost(positionsOptNodes, positionsGivenNodes):
    total_cost = 0

    for i in range(len(positionsOptNodes)):
        for j in range(len(positionsGivenNodes)):
            dist = np.exp(-0.3 * (np.linalg.norm(positionsOptNodes[i] - positionsGivenNodes[j]) **
2))
            total_cost += dist

    for i in range(len(positionsOptNodes)):
        for j in range(i + 1, len(positionsOptNodes)):
            dist = np.exp(-0.3 * (np.linalg.norm(positionsOptNodes[i] - positionsOptNodes[j]) ** 2))
            total_cost += dist

    return total_cost

# System of equations function for the place cost
def PlaceCost(x):
    return (x[0]**4 + x[1]**4) / 1000 + np.sin(x[0]) + np.cos(x[1]) / 5 + 0.4

# Define the city area limits
areaLim = 10

n = 10  # Number of existing stores
```

```
optNodes = 2  # Number of new stores to add

# Random positions of existing stores
positionsGivenNodes = np.random.uniform(-10, 10, (n, 2))
# Random positions of new stores
positionsOptNodes = np.random.uniform(-10, 10, (optNodes, 2))

def visualization(positionsGivenNodes, positionsOptNodes):
    plt.plot(positionsGivenNodes[:,0], positionsGivenNodes[:,1], 'or', label='Given Points',
linestyle='None')
    plt.plot(positionsOptNodes[:,0], positionsOptNodes[:,1], 'ob', label='New Points',
linestyle='None')
    plt.xlim([-areaLim-2, areaLim+2])
    plt.ylim([-areaLim-2, areaLim+2])
    plt.plot([-areaLim, -areaLim, areaLim, areaLim, -areaLim], [-areaLim, areaLim, areaLim, -
areaLim, -areaLim], '--k')
    plt.grid()
    plt.legend()
    plt.show()

visualization(positionsGivenNodes, positionsOptNodes)

# Function to calculate the distance between two points
def distanceBetweenTwoPoints(point1, point2):
    return np.linalg.norm(point1 - point2)

# Function to calculate the average distance between all points
def averageDistanceBetweenAllPoints(positionsGivenNodes, positionsOptNodes):
    fullDistance, totalEdges = 0, 0
    for i in range(len(positionsGivenNodes)):
        for j in range(len(positionsOptNodes)):
            fullDistance += distanceBetweenTwoPoints(positionsGivenNodes[i],
positionsOptNodes[j])
            totalEdges += 1

    for i in range(len(positionsOptNodes)):
        for j in range(i + 1, len(positionsOptNodes)):
            fullDistance += distanceBetweenTwoPoints(positionsOptNodes[i],
positionsOptNodes[j])
            totalEdges += 1
    return fullDistance / totalEdges

# Function to calculate the objective function
def objectiveFunction(positionsGivenNodes, positionsOptNodes):
    objFuncVal = 0
```

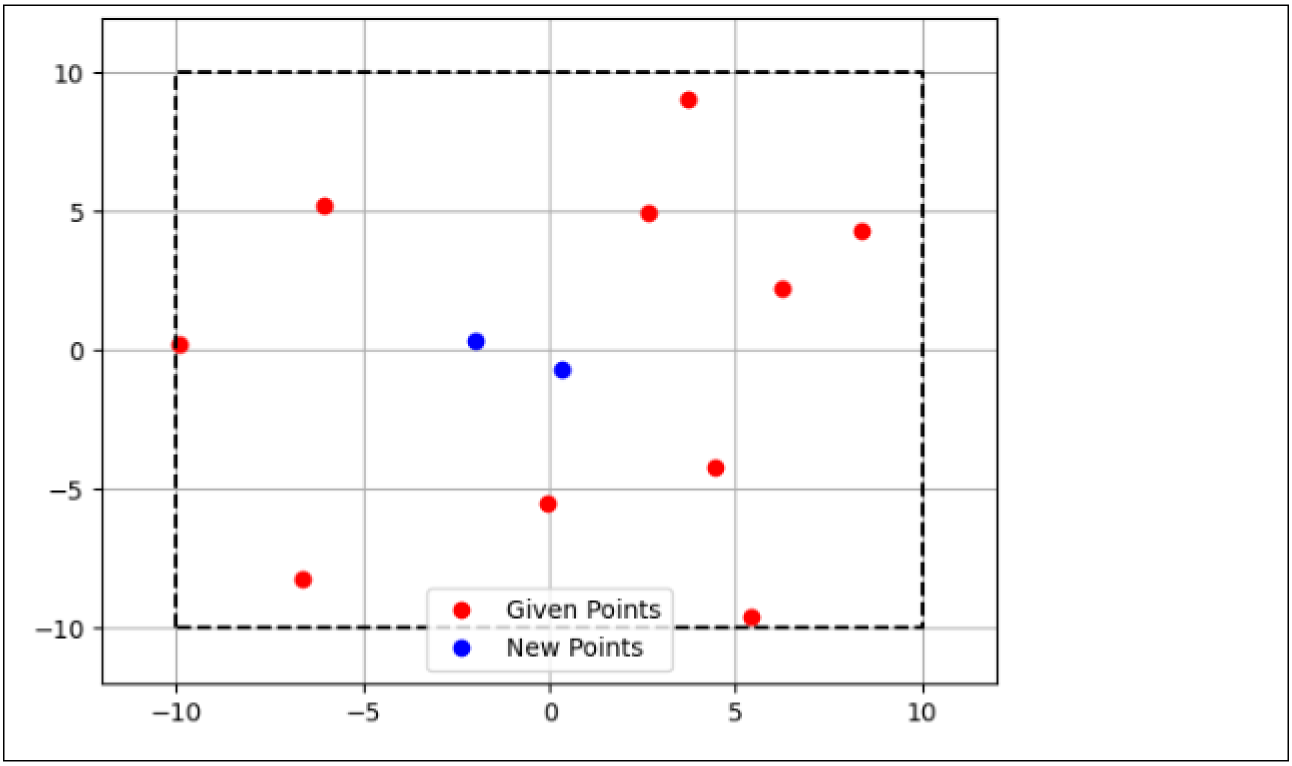
```
    avgDist = averageDistanceBetweenAllPoints(positionsGivenNodes, positionsOptNodes)

    for i in range(len(positionsGivenNodes)):
        for j in range(len(positionsOptNodes)):
            edgeDistance = distanceBetweenTwoPoints(positionsGivenNodes[i],
ositionsOptNodes[j])
            objFuncVal += (avgDist - edgeDistance) ** 2

    for i in range(len(positionsOptNodes)):
        for j in range(i + 1, len(positionsOptNodes)):
            edgeDistance = distanceBetweenTwoPoints(positionsOptNodes[i],
ositionsOptNodes[j])
            objFuncVal += (avgDist - edgeDistance) ** 2

    # Add cost to the objective function
    cost = SystemOfEquationsCost(positionsOptNodes, positionsGivenNodes)
    objFuncVal += cost

    return objFuncVal

# Function to calculate the quasi-gradient
lef quasiGradient(positionsGivenNodes, positionsOptNodes):
    h = 0.00001
    f0 = objectiveFunction(positionsGivenNodes, positionsOptNodes)
    df = positionsOptNodes * 0
    for i in range(len(positionsOptNodes)):
        for j in range(2):  # x and y coordinates
            positionsOptNodesNew = positionsOptNodes.copy()
            positionsOptNodesNew[i][j] += h
            f1 = objectiveFunction(positionsGivenNodes, positionsOptNodesNew)
            df[i][j] = (f1 - f0) / h
    return df

# Optimization loop
ter, step, eps = 0, 0.1, 1e-6
bjValOld = objectiveFunction(positionsGivenNodes, positionsOptNodes)
rint("Initial objective value: " + str(objValOld))
grad = quasiGradient(positionsGivenNodes, positionsOptNodes)

while np.linalg.norm(grad[:, :]) > eps and iter < 1000 and step > 1e-6:
    grad = grad / np.linalg.norm(grad[:, :])
    positionsOptNodes -= step * grad
    objValNew = objectiveFunction(positionsGivenNodes, positionsOptNodes)
    print(objValNew)
```

```
    if objValOld < objValNew:
        positionsOptNodes += step * grad
        step = step * 0.9
    else:
        objValOld = objValNew
    grad = quasiGradient(positionsGivenNodes, positionsOptNodes)
    iter += 1

print("After optimization: " + str(objValNew))
visualization(positionsGivenNodes, positionsOptNodes)
```

Rezultatai:



Initial objective value: 312.8119020286492
After optimization: 121.10919131068896


Given Points
New Points