Kauno technologijos universitetas

Informatikos fakultetas

II projektinio darbo ataskaita

Laboratorinio užduotis

Aistis Jakutonis

Studentas

Čalnerytė Dalia
Kriščiūnas Andrius

Dėstytojai

Kaunas, 2025

Turinys

1. Pirmas uždavinys

Užduoties variantas:

Metodai: LU arba Gauso (jei matrica singuliari)

Lygčių sistemos:

$$\begin{cases} 3x_1 + 10x_2 + x_3 + 5x_4 = 83 \\ -2x_1 + 6x_2 + 12x_3 + 14x_4 = 178 \\ 3x_1 + 12x_2 + 5x_3 + x_4 = 37 \\ -3x_1 - 9x_2 + 5x_3 = -26 \end{cases}$$

$$\begin{cases} 3x_1 + x_2 - x_3 + 5x_4 = 20 \\ -3x_1 + 4x_2 - 8x_3 - x_4 = -36 \\ x_1 - 3x_2 + 7x_3 + 6x_4 = 41 \\ 5x_2 - 9x_3 + 4x_4 = -16 \end{cases}$$

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

1.1. Tiesinių lygčių sistemų sprendimas

Gauso metodo formulės:

Eliminacija (žingsnis k):

$$m_{ik} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \quad i = k+1, \dots, n$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - m_{ik}\, a_{kj}^{(k-1)}, \quad i = k+1, \dots, n,\ j = k, \dots, n$$

$$b_i^{(k)} = b_i^{(k-1)} - m_{ik}\, b_k^{(k-1)}, \quad i = k+1, \dots, n$$

Atgalinė eiga:

$$x_n = \frac{b_n^{(n)}}{a_{nn}^{(n)}}$$

$$x_i = \frac{b_i^{(n)} - \sum_{j=i+1}^{n} a_{ij}^{(n)}\ x_j}{a_{ii}^{(n)}}, \quad i = n-1, \dots, 1$$

LU metodo formulės:

Išskaidymas A = LU:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}\ u_{kj}, \quad 1 \le i \le j \le n$$

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik}\ u_{kj}}{u_{jj}}, \quad 1 \le j < i \le n$$

Sprendimas Ax = b:

Ly = b:

$$y_i = b_i - \sum_{j=1}^{i-1} l_{ij}\ y_j, \quad i = 2, \dots, n$$

Ux = y:

$$x_i = \frac{y_i - \sum_{j=i+1}^{n} u_{ij}\ x_j}{u_{ii}}, \quad i = n-1, \dots, 1$$

1.1.1. Pirma tiesinių lygčių sistema

```
=== Sistema: Sistema #1 ===
Lygčių sistema:
3.00x1 + 10.00x2 + 1.00x3 + 5.00x4 = 83.00
-2.00x1 + 6.00x2 + 12.00x3 + 14.00x4 = 178.00
3.00x1 + 12.00x2 + 5.00x3 + 1.00x4 = 37.00
-3.00x1 - 9.00x2 + 5.00x3 + 0.00x4 = -26.00

Matrica A:
[[ 3. 10.  1.  5.]
 [-2.  6. 12. 14.]
 [ 3. 12.  5.  1.]
 [-3. -9.  5.  0.]]

Vektorius b:
[ 83. 178.  37. -26.]

LU Rezultatai:
[-2.  3. -1. 12.]

Ax = b
[0. 0. 0. 0.]
Rasti rezultatai yra pakankamai tikslūs (|Ax - b| ≤ 1e-9)

Rezultatai naudojant standartines funkcijas:
[-2.  3. -1. 12.]
```

1 pav. Pirmos lygčių sistemos sprendimo rezultatai

Buvo rastas vienintelis sprendinys ir jis buvo teisingas. Sprendinio tikslumas buvo įvertintas gautas vertes įstačius į lygtis ir atėmus iš reikiamų rezultatų, kadangi atėmus rezultatai buvo mažesni nei 1e-9 tai reiškia rezultatai yra pakankamai tikslūs, bei palyginus su standartinių funkcijų gautais rezultatais.

1.1.2. Antra tiesinių lygčių sistema

```
=== Sistema: Sistema #2 ===
Lygčių sistema:
3.00x1 + 1.00x2 - 1.00x3 + 5.00x4 = 20.00
-3.00x1 + 4.00x2 - 8.00x3 - 1.00x4 = -36.00
1.00x1 - 3.00x2 + 7.00x3 + 6.00x4 = 41.00
0.00x1 + 5.00x2 - 9.00x3 + 4.00x4 = -16.00

Matrica A:
[[ 3.  1. -1.  5.]
 [-3.  4. -8. -1.]
 [ 1. -3.  7.  6.]
 [ 0.  5. -9.  4.]]

Vektorius b:
[ 20. -36.  41. -16.]

LU nepavyko (matrica singuliari arba beveik singuliari). Pereiname prie Gauso metodo.
Išvada: be galo daug sprendinių.

Parinktas konkretus sprendinys:
[ 3.  18.5 12.5  1. ]

Ax = b
[0.00000000e+00 0.00000000e+00 2.13162821e-14 0.00000000e+00]
Rasti rezultatai yra pakankamai tikslūs (|Ax - b| ≤ 1e-9)

Rezultatai naudojant standartines funkcijas:
[ 3.         -1.00280636  2.51075772  2.90271282]
```

2 pav. Antros lygčių sistemos sprendimo rezultatai

Kadangi matrica buvo singuliari arba pakankamai singuliari, tai buvo pereita prie Gauso metodo. Buvo nustatyta, kad yra be galo daug sprendinių. Buvo parinktas konkretus sprendinys norint patikrinti teisingą sprendimą. Sprendinio tikslumas buvo įvertintas gautas vertes įstačius į lygtis ir atėmus iš reikiamų rezultatų, kadangi atėmus rezultatai buvo mažesni nei 1e-9 tai reiškia rezultatai yra pakankamai tikslūs. Tačiau matome, kad palyginti rezultatų su standartinėmis funkcijomis nepavyko, nes iš daugybės sprendinių variantų buvo gautas kitas sprendinys.

1.1.3. Trečia tiesinių lygčių sistema

```
=== Sistema: Sistema #3 ===
Lygčių sistema:
2.00x1 + 5.00x2 + 1.00x3 + 2.00x4 = -1.00
-2.00x1 + 0.00x2 + 3.00x3 + 5.00x4 = 7.00
1.00x1 + 0.00x2 - 1.00x3 + 1.00x4 = 3.00
0.00x1 + 5.00x2 + 4.00x3 + 7.00x4 = 4.00

Matrica A:
[[ 2.  5.  1.  2.]
 [-2.  0.  3.  5.]
 [ 1.  0. -1.  1.]
 [ 0.  5.  4.  7.]]

Vektorius b:
[-1.  7.  3.  4.]

LU nepavyko (matrica singuliari arba beveik singuliari). Pereiname prie Gauso metodo.
Išvada: sprendinių nėra.
```

3 pav. Trečios lygčių sistemos sprendimo rezultatai

Kadangi matrica buvo singuliari arba pakankamai singuliari, tai buvo pereita prie Gauso metodo. Buvo nustatyta, kad ši lygčių sistema sprendinių neturi.

Pagrindinės sprendimo dalies programos kodas:

```python
def gaussian_upper(A: np.ndarray, b: np.ndarray, tol: float = 1e-12) ->
np.ndarray:
    A = A.astype(float, copy=True)
    b = b.astype(float).reshape(-1, 1)
    #Isplestine matrica
    Ab = np.hstack([A, b])
    m, n_plus_1 = Ab.shape
    n = n_plus_1 - 1
    i = 0

    #Einame per stulpelius ir darome Gauso matrica
    for j in range(n):
        #Pasirenkame stulpelyje didžiausią pagal modulį elementą kaip
pagrindą
        pivot = i + np.argmax(np.abs(Ab[i:, j]))

        #Jei pivotas beveik nulis – stulpelyje nėra naudingos informacijos
        if np.abs(Ab[pivot, j]) < tol:
            continue

        #Jei reikia – sukeičiam eilutes, kad pivotas atsidurtų viršuje
        if pivot != i:
            Ab[[i, pivot], :] = Ab[[pivot, i], :]

        #Nuliname žemiau pivoto esančius elementus
        for r in range(i+1, m):
            if np.abs(Ab[r, j]) > tol:
                factor = Ab[r, j] / Ab[i, j]
                Ab[r, j:] -= factor * Ab[i, j:]

        #Pereiname prie kitos eilutes
        i += 1
```

```
        if i == m:
            break

    return Ab

def lu_solve(P: np.ndarray, L: np.ndarray, U: np.ndarray, b: np.ndarray) ->
np.ndarray:
    #Pritaikome permutaciją dešiniajam nariui: Pb = P * b
    Pb = P @ b
    y = np.zeros_like(b, dtype=float)
    x = np.zeros_like(b, dtype=float)

    #Išsprendžiame L y = P b (priekinei eigai nereikia dalybos iš įstrižainės,
nes L turi 1)
    for i in range(L.shape[0]):
        y[i] = Pb[i] - L[i, :i] @ y[:i]

    #Išsprendžiame U x = y (atgalinė eiga)
    for i in range(U.shape[0]-1, -1, -1):
        if abs(U[i, i]) < 1e-12:
            raise ValueError("Singuliarumo požymis.")

        x[i] = (y[i] - U[i, i+1:] @ x[i+1:]) / U[i, i]

    return x

def lu_factor_pp(A: np.ndarray, tol: float = 1e-12) -> Tuple[np.ndarray,
np.ndarray, np.ndarray]:
    #LU išskaidymas su daliniu perrinkimu: P*A = L*U
    A = A.astype(float).copy()
    n = A.shape[0]
    P = np.eye(n)
    L = np.zeros((n, n))
    U = A.copy()

    for k in range(n):
        #Randame didžiausią elementą stulpelyje k nuo eilutės k žemyn
        pivot = np.argmax(np.abs(U[k:, k])) + k

        #Jei pivotas ~0 – matrica singuliari ar beveik singuliari
        if abs(U[pivot, k]) < tol:
            raise ValueError("Matrica artima singuliariai.")

        #Perkeliame pivot eilutę į viršų ir atitinkamai atnaujiname P bei L
        if pivot != k:
            U[[k, pivot], :] = U[[pivot, k], :]
            P[[k, pivot], :] = P[[pivot, k], :]

            if k > 0:
                L[[k, pivot], :k] = L[[pivot, k], :k]

        #Skaičiuojame daugiklius (L) ir nuliname žemiau U įstrižainės
        for i in range(k+1, n):
            L[i, k] = U[i, k] / U[k, k]
            U[i, k:] = U[i, k:] - L[i, k] * U[k, k:]

    #Įrašome vienetus ant L įstrižainės
    np.fill_diagonal(L, 1.0)
```

```
    return P, L, U
```

2. Antras uždavinys

Užduoties variantas:

Metodas: Niutono

Lygčių sistema:

$$\begin{cases} \dfrac{x_1^2}{(x_2+\cos(x_1))^2+1}-2=0 \\ \left(\dfrac{x_1}{3}\right)^2+(x_2+\cos(x_1))^2-5=0 \end{cases}$$

Niutono metodo formulės:

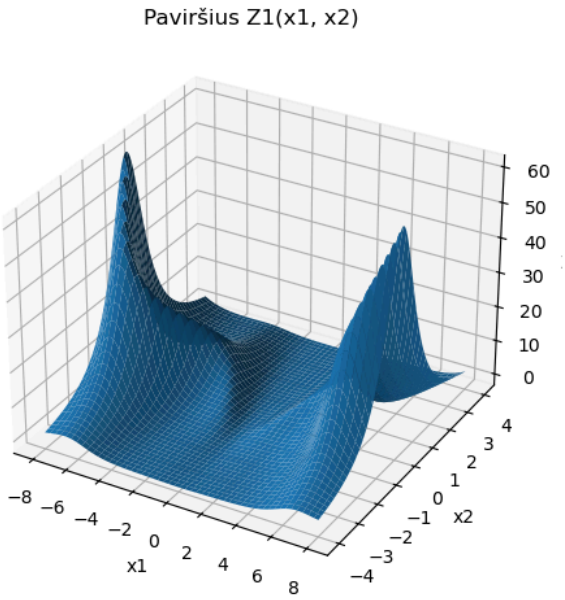
Jakobianas:

$$J_F(x)=\left[\frac{\partial F_i}{\partial x_j}(x)\right]_{i,j=1}^{n}$$

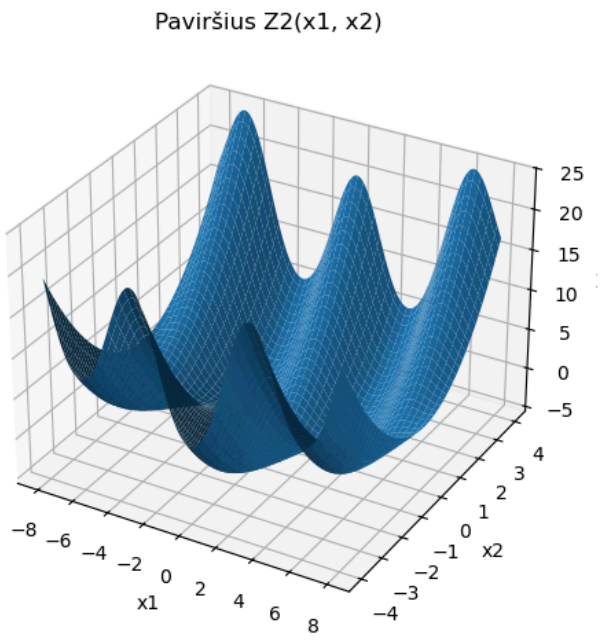
Iteracija:

$$x^{(k+1)}=x^{(k)}-J_F!\left(x^{(k)}\right)^{-1}F!\left(x^{(k)}\right),\quad k=0,1,2,$$

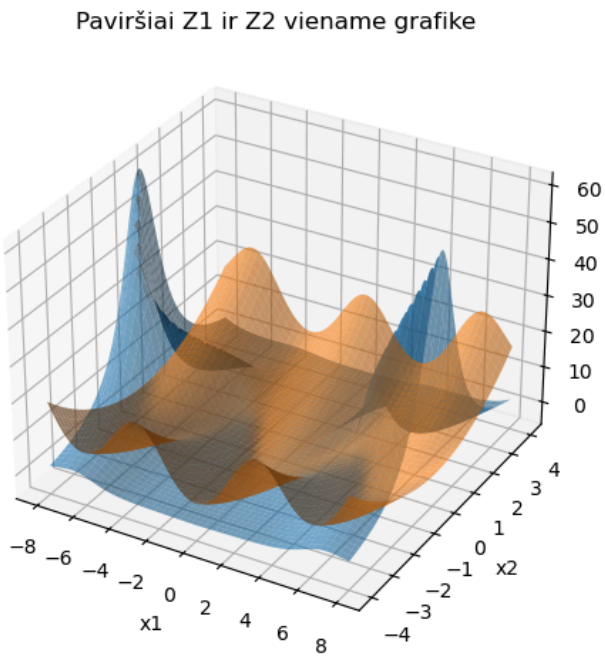
2.1. Dalis (a)



4 pav. Paviršius Z1

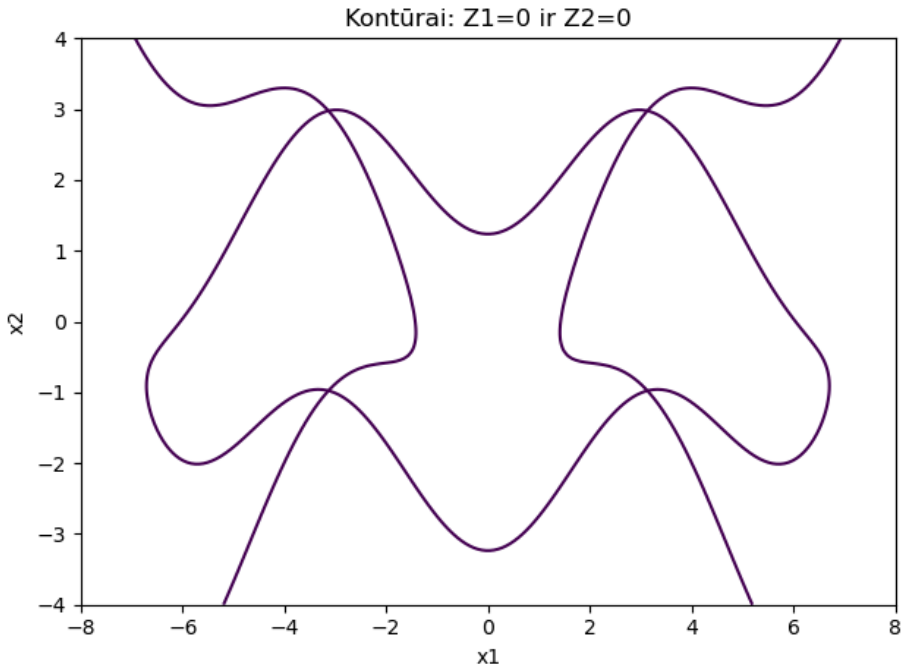


5 pav. Paviršius Z2



6 pav. Paviršiai Z1 ir Z2 viename grafike

2.2. Dalis (b)



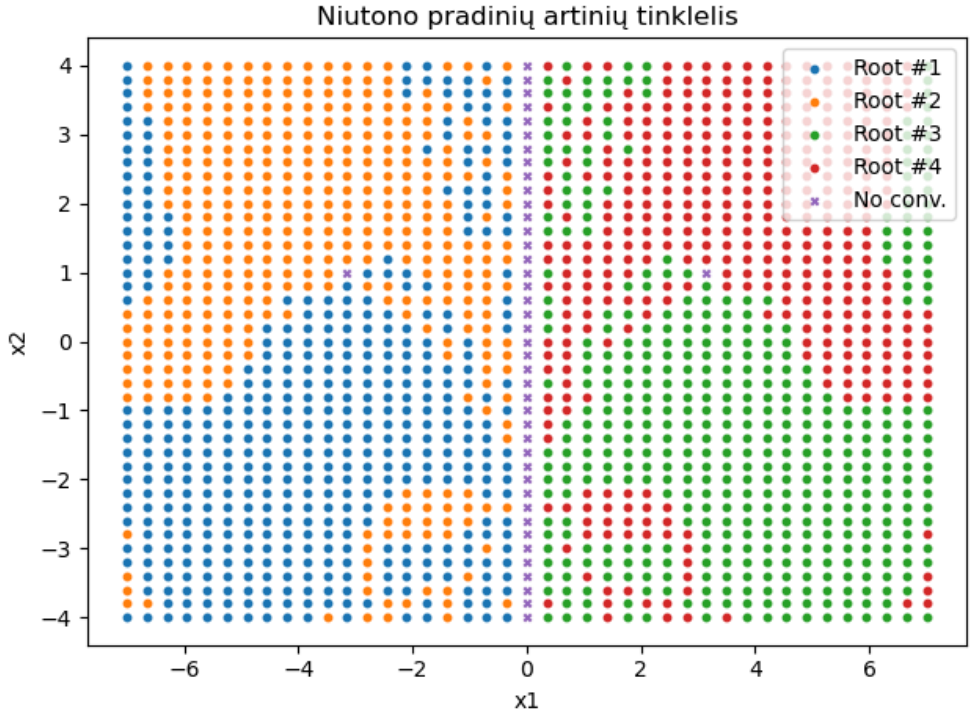
7 pav. Kontūrai Z1 = 0 ir Z2 = 0

Iš grafiko matyti, kad rezultatai yra maždaug tokie:

1 lentelė Rezultatai nustatyti iš grafiko

X	Y
-3,1	-1,2
-3,1	2,9
3,5	-1,2
3,5	2,9

2.3. Dalis (c)



8 pav. Pradinių artinių tinklelis

```
Rasta sprendinių: 4


Skirtingi sprendiniai ir bent po vieną pradinį artinį:

 Root #        x1        x2     ‖F(x)‖   Seed x1   Seed x2
      1 -3.133398 -0.977176  0.000000 -7.000000 -4.000000
      2 -3.133398  2.977109  0.000000 -7.000000 -3.800000
      3  3.133398 -0.977176  0.000000  0.350000 -4.000000
      4  3.133398  2.977109  0.000000  0.350000 -3.800000
```

9 pav. Sprendiniai su pradiniais artiniais

2 lentelė Sprendiniai ir pradiniai artiniai

Šaknys	x1	x2	||F(x)||	Pradinis x1	Pradinis x2
1	-3,133	-0,977	0	-7	-4
2	-3,133	2,977	0	-7	-3,8
3	3,133	-0,977	0	0,35	-4
4	3,133	2,977	0	0,35	-3,8

2.4. Dalis (d)

```
Sprendiniai (fsolve):

  Root #         x1         x2
       1   3.133398   2.977109
       2  -3.133398   2.977109
       3  -3.133398  -0.977176
       4   3.133398  -0.977176
```

10 pav. Sprendiniai rasti naudojant standartines funkcijas

Pagal standartines funkcijas, grafiką galima teigti, kad rasti sprendiniai buvo pakankamai tikslūs.

Pagrindinės sprendimo dalies programos kodas:

```python
def J(x: np.ndarray) -> np.ndarray:
    #Jakobiano matrica J(x) = [[dZ1/dx1, dZ1/dx2], [dZ2/dx1, dZ2/dx2]]
    x1, x2 = x[0], x[1]

    #Bendra daliklio dalis iš Z1 formulės
    denom = (x2 + np.cos(x1))**2 + 1.0
    #Išvestinės Z1 atžvilgiu x1 ir x2
    dZ1_dx1 = (2.0 * x1) / denom + (x1**2) * (2.0 * (x2 + np.cos(x1)) *
np.sin(x1)) / (denom**2)
    dZ1_dx2 = -(x1**2) * (2.0 * (x2 + np.cos(x1))) / (denom**2)

    #Išvestinės Z2 atžvilgiu x1 ir x2
    dZ2_dx1 = (2.0 * x1) / 9.0 - 2.0 * (x2 + np.cos(x1)) * np.sin(x1)
    dZ2_dx2 = 2.0 * (x2 + np.cos(x1))

    #Suformuojame 2x2 Jakobiano matricą
    return np.array([[dZ1_dx1, dZ1_dx2],
                     [dZ2_dx1, dZ2_dx2]], dtype=float)

def newton2(Ffun: Callable[[np.ndarray], np.ndarray],
            Jfun: Callable[[np.ndarray], np.ndarray],
            x0: np.ndarray,
            tol: float = 1e-9,
            max_iter: int = 60) -> Tuple[np.ndarray, bool, int]:

    #Niutono metodas 2D sistemai: iteratyviai ieško F(x)=0 sprendinio
    x = np.array(x0, dtype=float).copy()

    for k in range(1, max_iter + 1):
        #Apskaičiuojame F(x)
        f = Ffun(x)

        try:
            #Sprendžiame J(x) * step = F(x)
            step = np.linalg.solve(Jfun(x), f)
        except np.linalg.LinAlgError:
            #Jei Jakobianas singuliarus - blogai
            return x, False, k
```

```
        #Niutono atnaujinimas: x_{k+1} = x_k - step
        x_new = x - step

        #Sustabdymo kriterijai
        if np.linalg.norm(step, 2) < tol and np.linalg.norm(f, 2) < tol:
            return x_new, True, k

        #Tęsiame nuo naujos reikšmės
        x = x_new

    #Jei nepasiekta konvergencija per max_iter – grąžiname paskutinį x
    return x, False, max_iter
```

3. Trečias uždavinys

Užduotis:

Uždavinys: 1
Metodas: Greičiausio nusileidimo

5var.
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 ($\boldsymbol{x1}$, $\boldsymbol{y1}$) ir ($\boldsymbol{x2}$, $\boldsymbol{y2}$), kaina apskaičiuojama pagal formulę:

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

Parduotuvės, kurios koordinatės ($\boldsymbol{x1}$, $\boldsymbol{y1}$), vietos kaina apskaičiuojama pagal formulę:

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

Greičiausio nusileidimo metodo formulės:

Kryptis:

$$p^{(k)} = -\nabla f!\left(x^{(k)}\right)$$

Žingsnio dydis:

$$\alpha_k = \arg\min_{\alpha>0} f!\left(x^{(k)} + \alpha\, p^{(k)}\right)$$

Iteracijų atnaujinimas:

$$x^{(k+1)} = x^{(k)} + \alpha_k\, p^{(k)}$$

Gauta taškų konfigūracija:

```
n (esamos parduotuvės) = 6, m (naujos parduotuvės) = 3
```

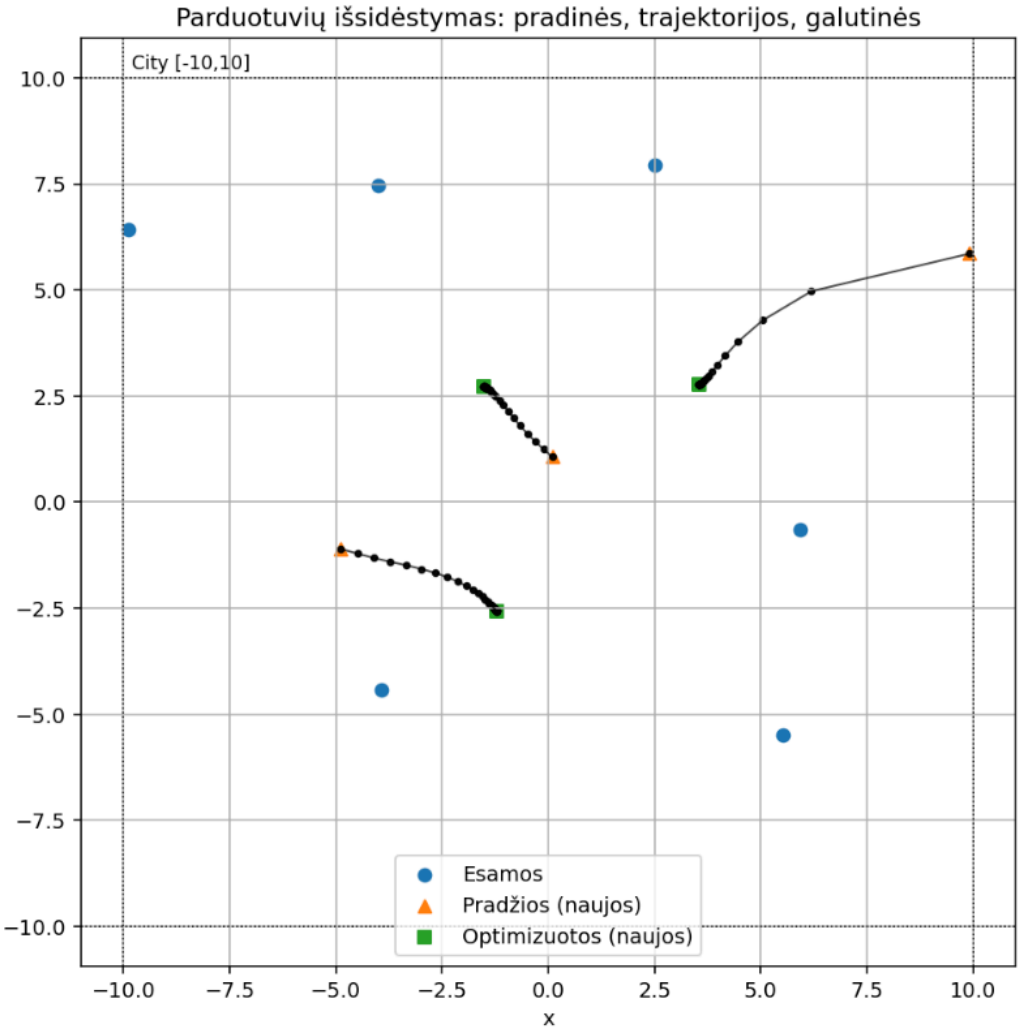
```
=== Esamos parduotuvės (x, y) ===
         x         y
 2.501909  7.944276
 5.513714 -5.495856
-3.996674  7.471069
-9.894694  6.424568
 5.941389 -0.641301
-3.939351 -4.431488

=== Pradinės naujų (x, y) ===
         x         y
-4.902608 -1.098474
 0.090965  1.069947
 9.910006  5.853238

=== Optimizuotos naujų (x, y) ===
         x         y
-1.217077 -2.563393
-1.508120  2.723003
 3.536341  2.762888
```

11 pav. Esamų ir gautų parduotuvių koordinatės

Pradinė ir gauta taškų konfigūracija:



12 pav. Pradinė ir gauta taškų konfigūracija

Taikyta tikslo funkcija:

```
Taikyta tikslo funkcija:
F = sum_j Cp(p_j) + sum_j sum_i exp(-0.3 ||p_j - q_i||^2) + sum_{j<k} exp(-0.3 ||p_j - p_k||^2)
Cp(x,y) = (x^4 + y^4)/1000 + (sin x + cos y)/5 + 0.4
```

13 pav. tikslo funkcija

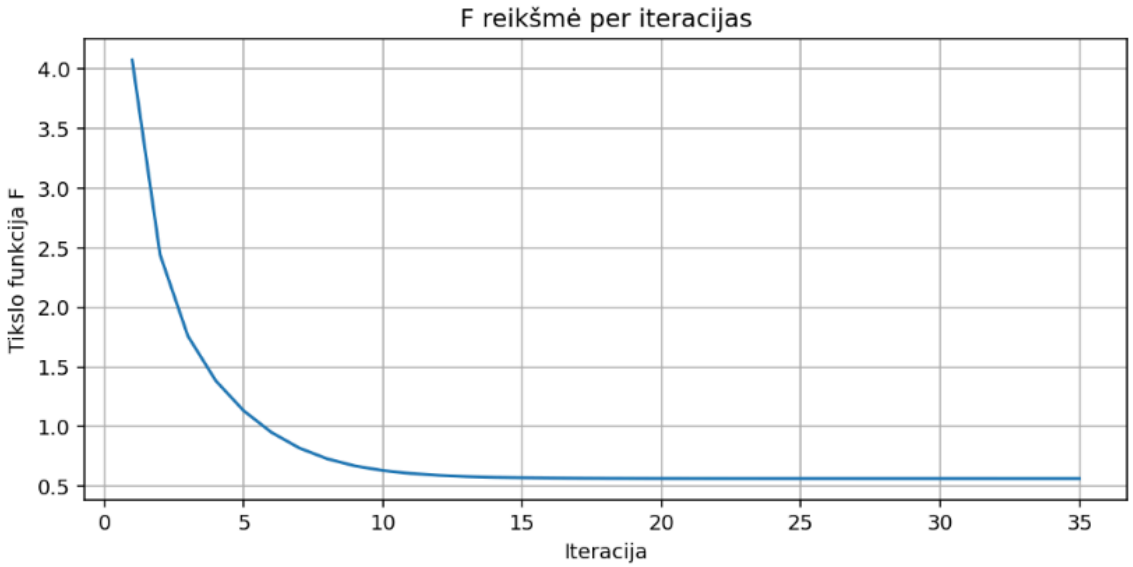
Parametrai:

Pradinio žingsnio dydis = 1
Žingsnio mažinimo koeficientas = 0,6
Minimalus mažėjimo reikalavimo koeficientas = 1e-4
Sustabdymo kriterijus = 1e-6

Iteracijų skaičius: 35

Iteracijų pabaigos sąlygos: 3000 iteracijų

Tikslo funkcijos priklausomybės nuo iteracijų skaičiaus grafikas:



14 pav. Tikslo funkcijos priklausomybės nuo iteracijų skaičiaus grafikas

Pagrindinės sprendimo dalies programos kodas:

```
#Bendra tikslo funkcija F
#Įėjimas: 'flat_new' - vektorius [x1,y1,...,xm,ym], 'existing' - esamų
parduotuvių masyvas
#Išėjimas: skaliarinė F reikšmė
def objective(flat_new, existing):
    P = flat_new.reshape(-1, 2)

    #Vietos kainų suma
    F = np.sum([Cp(p) for p in P])

    #Porinės kainos tarp naujų ir esamų
```

```
    for p in P:
        F += np.sum([C_pair(p, q) for q in existing])

    #Porinės kainos tarp pačių naujų (kiekvieną porą skaičiuojame 1 kartą:
j<k)
    for j in range(len(P)):
        for k in range(j + 1, len(P)):
            F += C_pair(P[j], P[k])

    return float(F)

#Analitinis F gradientas pagal naujų parduotuvių koordinates
#Grąžina tokios pačios formos vektorių kaip 'flat_new'
def gradient(flat_new, existing):
    P = flat_new.reshape(-1, 2)
    grad = np.zeros_like(P)

    #Vietos kainos Cp išvestinės
    grad[:, 0] += 4.0 * P[:, 0] ** 3 / 1000.0 + np.cos(P[:, 0]) / 5.0
    grad[:, 1] += 4.0 * P[:, 1] ** 3 / 1000.0 - np.sin(P[:, 1]) / 5.0

    #Porinės kainos su esamomis parduotuvėmis
    for j, p in enumerate(P):
        diffs = p - existing
        d2 = np.sum(diffs**2, axis=1)
        e = np.exp(-0.3 * d2)
        grad[j] += np.sum(((-0.6) * e)[:, None] * diffs, axis=0)

    #Porinės kainos tarp pačių naujų: kiekvieną neordinuotą porą skaičiuojame
kartą (j<k)
    m_pts = len(P)
    for j in range(m_pts):
        for k in range(j + 1, m_pts):
            diff = P[j] - P[k]
            d2 = np.dot(diff, diff)
            e = np.exp(-0.3 * d2)
            gpair = (-0.6) * e * diff      # d/d p_j
            grad[j] += gpair
            grad[k] -= gpair               # d/d p_k = -gpair
            diff = P[j] - P[k]
            d2 = np.dot(diff, diff)
            e = np.exp(-0.3 * d2)
            if j < k:
                grad[j] += (-0.6) * e * diff

    return grad.reshape(-1)

#Greičiausio nusileidimo metodas (Armijo žingsnis)
#'existing' - esamų parduotuvių masyvas
#x0 - pradinės naujų parduotuvių koordinatės (m,2)
#Grąžina: optimizuotą vektorių,
#sąrašą su f, žingsniu ir ||grad||,
#trajektoriją kiekvienai naujai parduotuvei (taškai per iteracijas)
def steepest_descent(existing, x0, max_iter=2000, tol=1e-6,
                     alpha0=1.0, beta=0.5, sigma=1e-4, track_every=1):
    x = x0.reshape(-1).astype(float)
    m = x.size // 2
```

```
    #Trajektorijos kaupimas: kiekvienai naujai parduotuvei saugom visi priimti
taškai
    paths = [[x[2*j:2*j+2].copy()] for j in range(m)]
    hist = []

    f = objective(x, existing)
    for it in range(1, max_iter + 1):
        g = gradient(x, existing)
        gnorm2 = np.dot(g, g)

        #Sustabdymas pagal mažą gradiento normą
        if np.sqrt(gnorm2) < tol:
            hist.append({"iter": it, "f": f, "alpha": 0.0, "gnorm":
np.sqrt(gnorm2), "stop": "||grad||<tol"})
            break

        #Armijo backtracking: ieškomas žingsnis alpha, tenkinantis F(x-
alpha*g) <= F(x) - sigma*alpha*||g||^2
        alpha = alpha0
        dirdec = sigma * alpha * gnorm2
        while True:
            x_new = x - alpha * g
            f_new = objective(x_new, existing)
            if f_new <= f - sigma * alpha * gnorm2:   # <-- čia svarbu α
kiekvieną kartą
                break
            alpha *= beta
            #žingsnis tapo per mažas – laikome, kad progresas neįmanomas
            if alpha < 1e-12:
                hist.append({"iter": it, "f": f, "alpha": alpha, "gnorm":
np.sqrt(gnorm2), "stop": "alpha too small"})
                return x, hist, paths

        #Patvirtiname žingsnį
        x = x_new
        f = f_new

        #Užfiksuojame trajektorijų taškus ir istoriją
        if it % track_every == 0:
            for j in range(m):
                paths[j].append(x[2*j:2*j+2].copy())
            hist.append({"iter": it, "f": f, "alpha": alpha, "gnorm":
np.sqrt(gnorm2)})

        #Papildomas sustabdymas: labai maža santykinė pažanga
        if len(hist) >= 2:
            prevf = hist[-2]["f"]
            if abs(prevf - f) <= max(1.0, abs(prevf)) * 1e-10:
                hist.append({"iter": it, "f": f, "alpha": alpha, "gnorm":
np.sqrt(gnorm2), "stop": "rel. change tiny"})
                break

    else:
        #pasiekta max_iter
        hist.append({"iter": max_iter, "f": f, "alpha": 0.0,
                     "gnorm": float(np.linalg.norm(gradient(x, existing))),
"stop": "max_iter"})

    return x, hist, paths
```