VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMACINIŲ SISTEMŲ INŽINERIJOS STUDIJŲ PROGRAMA

Optimizavimas be apribojimų

Laboratorinis darbas

Atliko:	2 kurso 1 grupės studentas
	Igor Repkin
Darbo vadovas:	prof. dr. Julius Žilinskas

Vilnius - 2023

TURINYS

1. Įvadas

Šiame laboratoriniame darbe reikia suprogramuoti gradientinio nusileidimo, greičiausiojo nusileidimo ir deformuojamo simplekso algoritmus. Suformuluoti ir minimizuoti uždavinį, naudojant suprogramuotus algoritmus pradedant taškuose $X_0 = (0{,}0)$, $X_1 = (1, 1)$, $X_m = (\frac{a}{10}, \frac{b}{10})$, kur $a = 6$ ir $b = 9$ bei surasti kokie turėtų būti stačiakampio gretasienio formos dėžės matmenys, kad vienetiniam paviršiaus plotui jos tūris būtų maksimalus.

2. Užduotys

2.1. Užduoties formulavimas

1. Vienetinio dėžės paviršiaus ploto reikalavimas ir vieno kintamojo išraiška per kitus:

$$x_1 + x_2 + x_3 = 1$$

$$x_3 = 1 - x_1 - x_2$$

   x_1, x_2, x_3 - priekinės ir galinės sienų plotų suma, šoninių sienų plotų suma, viršutinės ir apatinės sienų plotų suma.

2. Dėžės tūrio pakelto kvadratu (tikslo) funkcija:

$$V^2 = (abc)^2$$

$$V^2 = ab * bc * ac$$

$$ab = \frac{x_1}{2}, bc = \frac{x_2}{2}, ac = \frac{x_3}{2}$$

$$V^2 = \frac{x_1 x_2 x_3}{8}$$

$$f(x) = V^2 = \frac{x_1 x_2 (1 - x_1 - x_2)}{8}$$

   x_1, x_2, x_3 - priekinės ir galinės sienų plotų suma, šoninių sienų plotų suma, viršutinės ir apatinės sienų plotų suma ir a, b, c - dėžes kraštinių ilgiai.

3. Kad galėtume skaičiuoti tikslo funkcijos minimumą ją reikia padauginti iš -1. Taigi:

$$f(x) = -\frac{x_1 x_2 (1 - x_1 - x_2)}{8}$$

4. Funkcijos gradientas:

$$\frac{\partial}{\partial\, x_1}\left(-\frac{x_1 x_2\,(1\ -\ x_1\ -\ x_2)}{8}\right) = -\frac{x_2\,(-2x_1 - x_2 + 1)}{8}$$

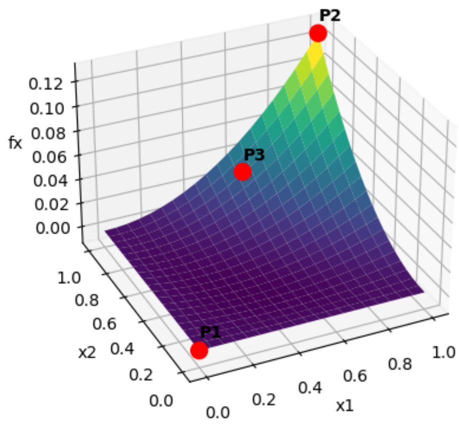
$$\frac{\partial}{\partial\, x_2}\left(-\frac{x_1 x_2\,(1\ -\ x_1\ -\ x_2)}{8}\right) = -\frac{x_1\,(1 - x_1 - 2x_2)}{8}$$

$$\nabla f(X) = \left(-\frac{x_2\,(-2x_1 - x_2 + 1)}{8}, -\frac{x_1\,(1 - x_1 - 2x_2)}{8}\right)$$

5. Tikslo ir gradiento funkcijų reikšmės pradiniuose taškuose:

1 lentelė. $f(X)$ ir $\nabla f(X)$ reikšmės pradiniuose taškuose.

Taškas	$f(X)$	$\nabla f(X)$
$(0, 0)$	0	(0.0, 0.0)
$(1, 1)$	0.125	(0.25, 0.25)
$(\frac{6}{10}, \frac{9}{10})$	0.03375	(0.12375, 0.105)



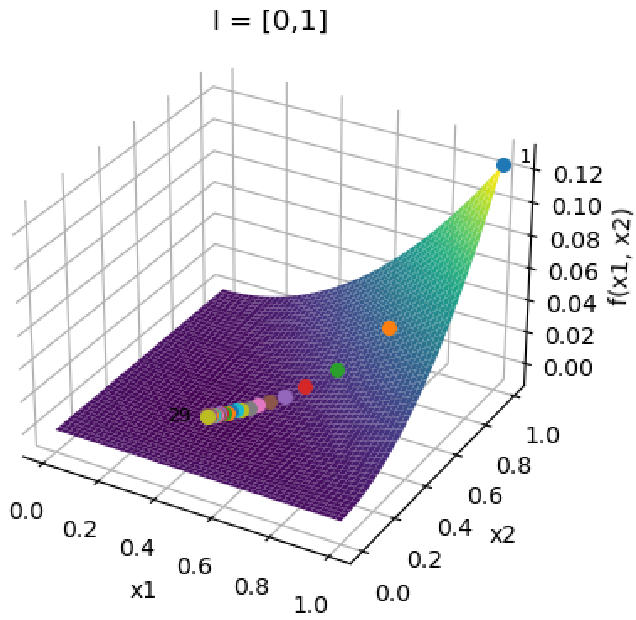
1 pav. Tikslo funkcijos vizualizavimas bei pradiniai taškai.

2.2. Gradientinio nusileidimo algoritmas

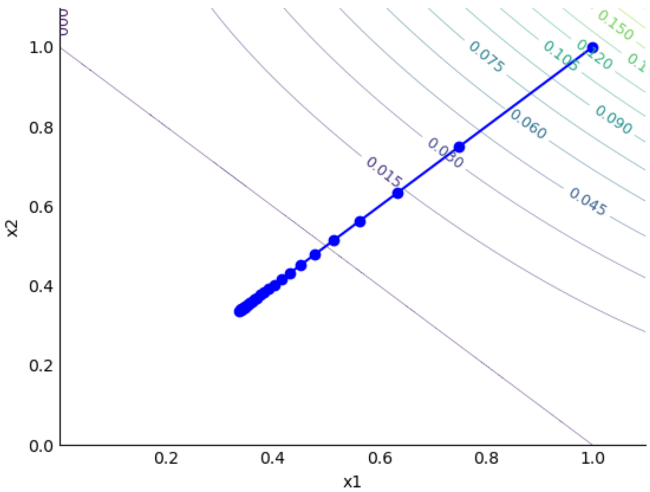
Su Python programavimo kalba buvo suprogramuotas gradientinio nusileidimo algoritmas (žr. 1 pr.). Algoritmas buvo paleistas iš 3 taškų: $(1, 1)$, $(0, 0)$, $(0.6, 0.9)$. Pagal algoritmo veikimo rezultatus buvo sudaryti grafikai (žr. 2 - 6 pav.). Algoritmo veikimo rezultatai buvo surašyti į lentelę (žr. 2 lent.). Kai pradinis taškas yra $(0, 0)$ algoritmas negali susirasti minimumo, nes tame taške funkcijos gradientas yra lygus nuliui. Kai pradinis taškas yra $(1, 1)$ algoritmas padaro 28 iteracijas ir artėja tiesiai prie minimumo. Kai pradinis taškas yra $(0.6, 0.9)$ algoritmui jau reikia padaryti daugiau žingsnių.

2 lentelė. Gradientinio nusileidimo algoritmo veikimo rezultatai.

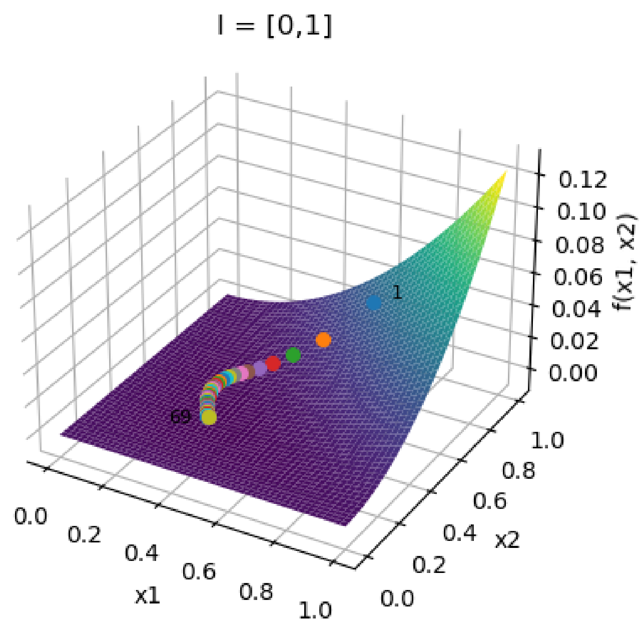
Taškas	$f(X)$ išk. sk.	Iteracijų sk.
$(0, 0)$	2	1
$(1, 1)$	56	28
$(\frac{6}{10}, \frac{9}{10})$	136	68



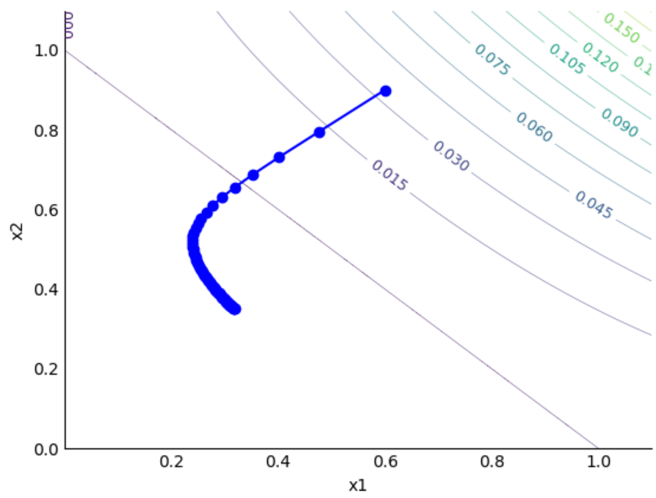
2 pav. Gradientinio nusileidimo algoritmo 3d vizualizacija. Pradinis taškas $[1, 1]$



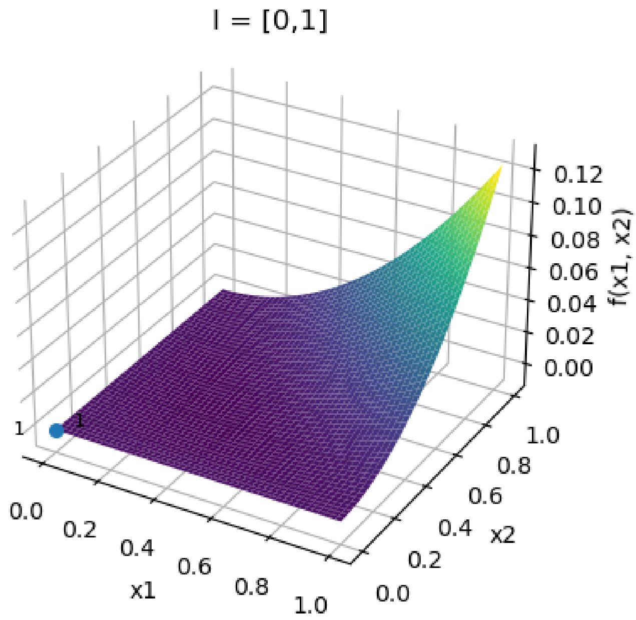
3 pav. Gradientinio nusileidimo algoritmo kontūrinė vizualizacija. Pradinis taškas $[1, 1]$



4 pav. Gradientinio nusileidimo algoritmo 3d vizualizacija. Pradinis taškas $[0.6, 0.9]$



5 pav. Gradientinio nusileidimo algoritmo kontūrinė vizualizacija. Pradinis taškas $[0.6, 0.9]$



6 pav. Gradientinio nusileidimo algoritmo 3d vizualizacija. Pradinis taškas $[0, 0]$

2.3. Greičiausiojo nusileidimo algoritmas

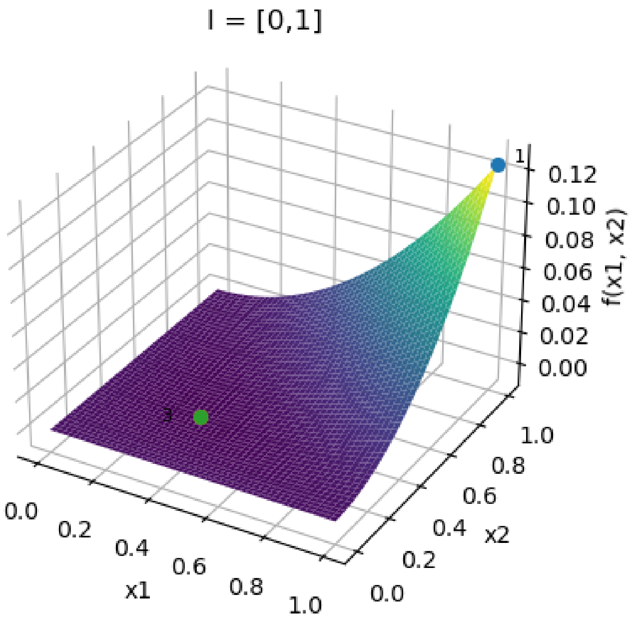
Greičiausiojo nusileidimo algoritmas yra labai panašus į gradientinio nusileidimo algoritmą, bet greičiausiojo nusileidimo algoritmas sprendžia papildomą optimizavimo uždavinį (mano implementacijoje tai Auksinio pjūvio alogritmas) (žr. 3 lent. ir 4 lent.) ir tam, kad nustatyti geriausią žingsnio daugiklį. Greičiausiojo nusileidimo algoritmas taip pat negali rasti funkcijos minimumo iš taško $(0, 0)$, nes tame taške funkcijos gradientas yra lygus nuliui (žr. 11 pav.). Lyginant su gradientinio nusileidimo algoritmu, greičiausiojo nusileidimo algoritmas labai greitai randa minimumą pradedant iš taško $(1, 1)$ (žr. 7 pav. ir 8 pav.). Bet sprendžiant uždavinį pradedant taške $(0.6, 0.9)$ algoritmas jau turi atlikti žymiai daugiau iteracijų bei panaudoti daug tikslo funkcijos iškvietimų papildomų uždavinių sprendimui (žr. 9 pav. ir 10 pav.).

3 lentelė. Greičiausiojo nusileidimo algoritmo veikimo rezultatai.

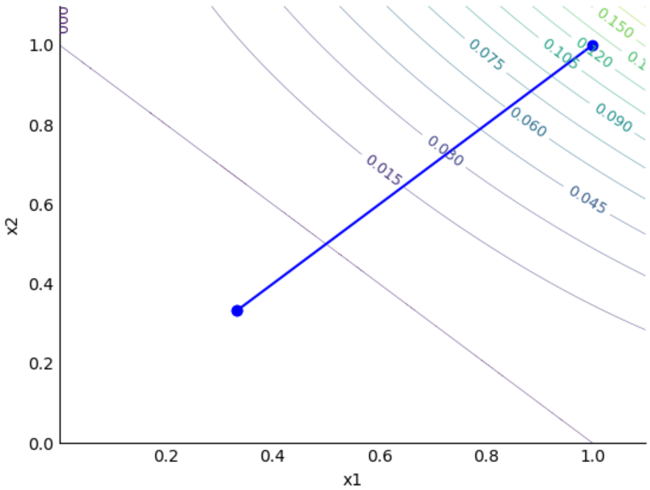
Taškas	$f(X)$ išk. sk.	Iteracijų sk.
$(0, 0)$	2	1
$(1, 1)$	4	2
$(\frac{6}{10}, \frac{9}{10})$	40	20

4 lentelė. Greičiausiojo nusileidimo papildomų uždavinių sprendimo statistika.

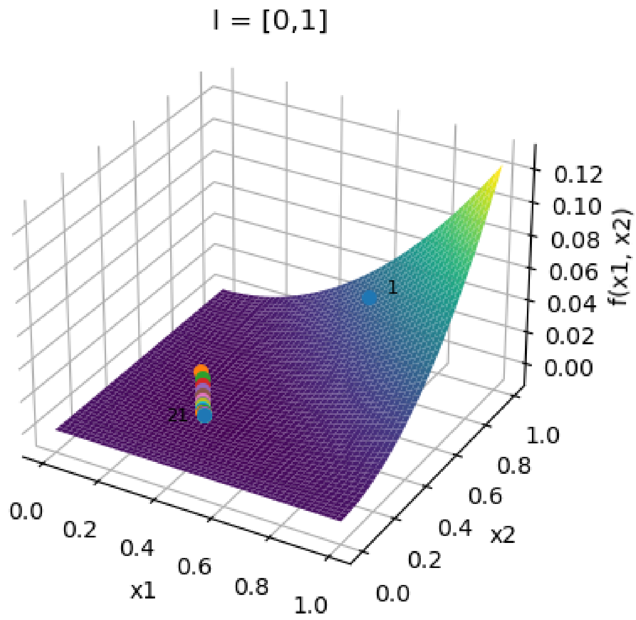
Taškas	$f(X)$ išk. sk.	Iteracijų sk.
$(0, 0)$	20	18
$(1, 1)$	40	36
$(\frac{6}{10}, \frac{9}{10})$	400	360



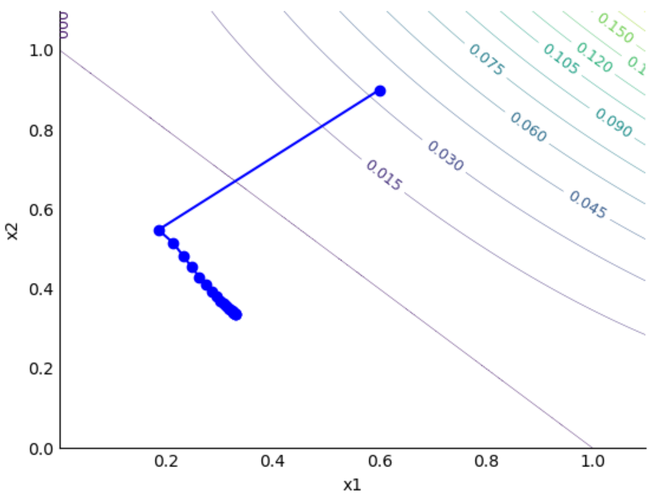
7 pav. Greičiausiojo nusileidimo algoritmo 3d vizualizacija. Pradinis taškas $[1, 1]$



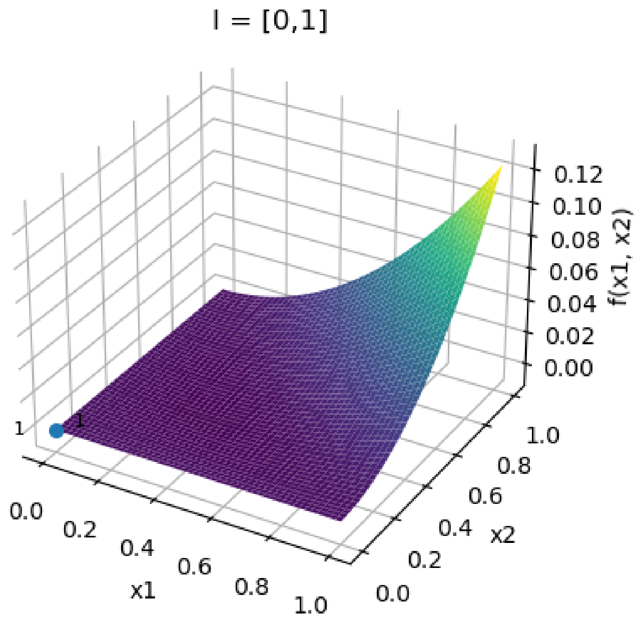
8 pav. Greičiausiojo nusileidimo algoritmo kontūrinė vizualizacija. Pradinis taškas $[1, 1]$



9 pav. Greičiausiojo nusileidimo algoritmo 3d vizualizacija. Pradinis taškas $[0.6, 0.9]$



10 pav. Greičiausiojo nusileidimo algoritmo kontūrinė vizualizacija. Pradinis taškas $[0.6, 0.9]$



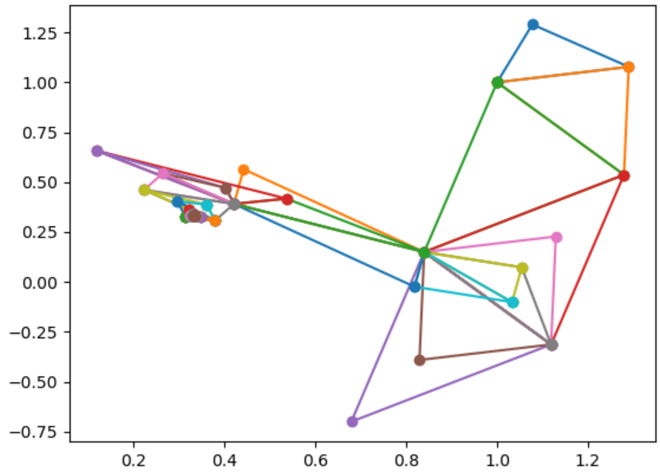
11 pav. Greičiausiojo nusileidimo algoritmo 3d vizualizacija. Pradinis taškas $[0, 0]$

2.4. Deformuojamo simplekso algoritmas

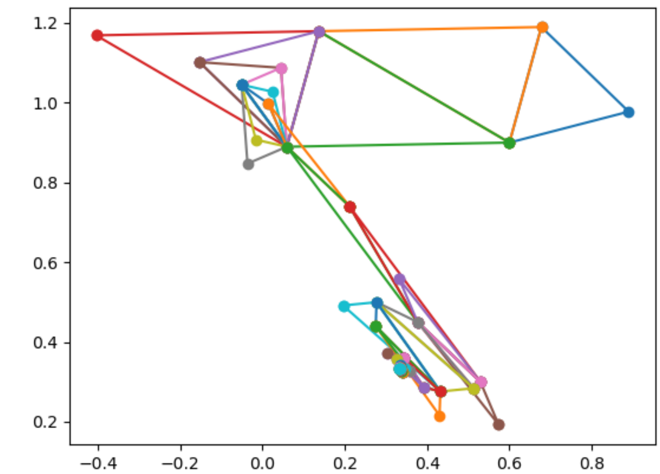
Deformuojamo simplekso algoritmas nenaudoja funkcijos gradiento skaičiavimams ir dėl to jis gali surasti funkcijos minimumą net ir iš taško, kur funkcijos gradientas yra lygus nuliui (žr. 14 pav.). Algoritmo veikimo rezultatai yra atvaizduoti 5 lentelėje. Grafikai atvaizduoja visas algoritmo iteracijas nuo pirmos iki paskutinės. Kiekvienos iteracijos taškai sudaro tam tikros spalvos trikampį ant grafiko (žr. 12 pav., 13 pav., 14 pav.). Deformuojamo simplekso algoritmo realizacija galima peržiūrėti 3 priede.

5 lentelė. Deformuojamo simplekso algoritmo veikimo rezultatai.

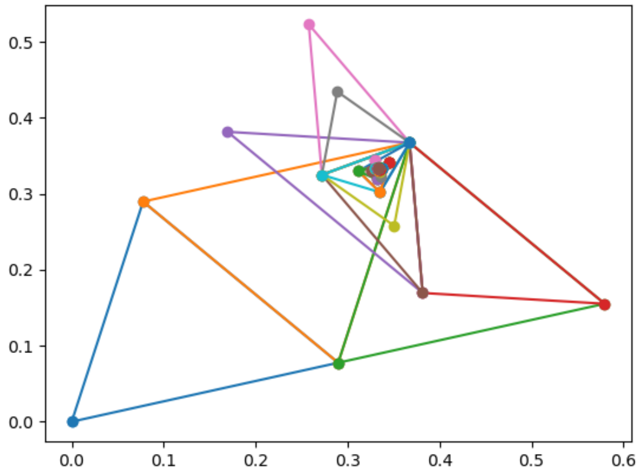
Taškas	$f(X)$ išk. sk.	Iteracijų sk.
$(0, 0)$	71	25
$(1, 1)$	101	35
$(\frac{6}{10}, \frac{9}{10})$	111	39



12 pav. Deformuojamo simplekso algoritmo vizualizacija pradiniame taške $[1, 1]$



13 pav. Deformuojamo simplekso algoritmo vizualizacija pradiniame taške $[0.6, 0.9]$



14 pav. Deformuojamo simplekso algoritmo vizualizacija pradiniame taške $[0, 0]$

3. Rezultatai, palyginimas ir išvados

Lyginant rezultatus galima pamatyti, kad gradientinio ir greičiausiojo nusileidimo metodai negalėjo rasti funkcijos minimumo, kai gradientas yra lygus nuliui (žr. 6 pav. ir 11 pav.). Greičiausiojo nusileidimo metodo efektyvumas greitai mažėja, nes jis reikalauja, kad būtų sprendžiamas papildomas optimizavimo uždavinys, bet yra atvejų, kai jis gali surasti minimumą vos per 2 iteracijas (žr. 6 lent.). Deformuojamo simplekso algoritmas galėjo rasti minimumo tašką visuose atvejuose. Po minimizavimo užduoties atlikimo, išsiaiškinau, kad optimali dėžės forma (kad vienetiniam paviršiaus plotui jos tūris būtų maksimalus) – *kubas*.

6 lentelė. Rezultatų palyginimas taške $(1, 1)$.

Pavadinimas	$f(X)$ išk. sk.	It. sk.	Atrastas taškas	Funk. min. įvertis
Gradientinio nus. alg.	56	28	$(0.33785945, 0.33785945)$	-0.0046296257
Greičiausiojo nus.	4 + 40	2 + 36	$(0.33331962, 0.33331962)$	-0.0046296129
Deformuojamo sim. alg.	101	35	$(0.33307098, 0.33326170)$	-0.0046296265

7 lentelė. Rezultatų palyginimas taške $(0.6, 0.9)$.

Pavadinimas	$f(X)$ išk. sk.	It. sk.	Atrastas taškas	Funk. min. įvertis
Gradientinio nus. alg.	136	68	$(0.31797040, 0.35005732)$	-0.00461020311
Greičiausiojo nus.	40 + 400	20 + 360	$(0.33066412, 0.33603833)$	-0.00462962522
Deformuojamo s. alg.	111	39	$(0.33303561, 0.33328346)$	-0.00462961872

8 lentelė. Rezultatų palyginimas taške $(0, 0)$.

Pavadinimas	$f(X)$ išk. sk.	It. sk.	Atrastas taškas	Funk. min. įvertis
Gradientinio nus. alg.	2	1	$(0, 0)$	0
Greičiausiojo nus.	2 + 20	1 + 18	$(0, 0)$	0
Deformuojamo simp. alg.	71	25	$(0.33313146, 0.33337016)$	-0.004629612957

4. Priedai

```python
def gradient_descent(gradf, start, learning_rate=1, tolerance=0.001):
steps = [start]  # stat tracing
function_uses = 0
xi = start

while True:
    gradxi = gradf(xi)  # Find gradient at point X_i
    function_uses += 2

    xi = xi - learning_rate * gradxi  # Find X_i+1 = X_i - gamma * gradf(X_i)
    steps.append(xi)  # stat tracing

    if np.linalg.norm(learning_rate * gradxi) < tolerance:
        break
return steps, xi, function_uses
```

Listing 1: Gradientinio nusileidimo algoritmo realizacija su Python

```python
def golden_section(xi, gradxi, func, l=0, r=5, deltax=0.001):
tau = (-1 + math.sqrt(5)) / 2
stats = {"function_uses": 0, "iterations": 0}

# Step 1
L = r - l
point1 = r - tau * L
value_of_function_at_point1 = func(xi + point1 * (-gradxi))
stats["function_uses"] += 1
point2 = l + tau * L
value_of_function_at_point2 = func(xi + point2 * (-gradxi))
stats["function_uses"] += 1

while True:
    stats["iterations"] += 1
    # Step 2
    if value_of_function_at_point2 < value_of_function_at_point1:
        l = point1
        L = r - l
        point1 = point2
        value_of_function_at_point1 = value_of_function_at_point2

        point2 = l + tau * L
        value_of_function_at_point2 = func(xi + point2 * (-gradxi))
        stats["function_uses"] += 1
    # Step 3
    else:
        r = point2
        L = r - l
```

```
            point2 = point1
            value_of_function_at_point2 = value_of_function_at_point1

            point1 = r - tau * L
            value_of_function_at_point1 = func(xi + point1 * (-gradxi))
            stats["function_uses"] += 1
        # Step 4
        if L < deltax:
            return (point1 + point2) / 2, stats


def steepest_descent(f, gradf, start, tolerance=0.001):
    steps = [start]  # stat tracing
    stats_additional = {"function_uses": 0, "iterations": 0, "count": 0}
    function_uses = 0
    xi = start

    while True:
        gradxi = gradf(xi)  # Find gradient at point X_i
        function_uses += 2

        learning_rate, stats = golden_section(
            xi, gradxi, f
        )  # Find gamma such: arg min_gamma >= 0 f(X_i - gamma * gradf(X_i))

        stats_additional["function_uses"] += stats["function_uses"]
        stats_additional["iterations"] += stats["iterations"]
        stats_additional["count"] += 1

        xi = xi - learning_rate * gradxi  # Find X_i+1 = X_i - gamma * gradf(X_i
    )
        steps.append(xi)  # stat tracing

        if np.linalg.norm(learning_rate * gradxi) < tolerance:
            break
    return steps, xi, function_uses, stats_additional
```

Listing 2: Greičiausiojo nusileidimo algoritmo realizacija su Python

```
  def generate_points(
    f,
    starting_point,
    alpha=0.3
):  # Alpha is basically the length of the side of the initial simplex
    x0 = np.array(starting_point)
    n = len(x0)
    X = [x0]
    sqrt2 = math.sqrt(2)

    for i in range(n):
```

```
        si = []
        for j in range(n):
            if i == j:
                si.append((math.sqrt(n + 1) - 1) / (n * sqrt2) * alpha)
            else:
                si.append((math.sqrt(n + 1) + n - 1) / (n * sqrt2) * alpha)
        X.append(x0 + si)

    # Form a list of dictionaries with point coordinates and the value of a
   function
    return [{
        "coords": np.array(x[0]),
        "value": x[1]
    } for x in zip(X, [f(x) for x in X])]


def find_centroid(f, points, n, worst_points_index):
    centroid_coords = 1 / n * sum(
        [v['coords'] for i, v in enumerate(points) if i != worst_points_index
   ])
    return {"coords": centroid_coords, "value": f(centroid_coords)}


def step(f, points, centroid, worst_points_index, alpha):
    new_point_coords = centroid['coords'] + alpha * (
        centroid['coords'] - points[worst_points_index]['coords'])
    return {"coords": new_point_coords, "value": f(new_point_coords)}


def find_worst_points_index(points):
    return np.array([point['value'] for point in points]).argmax()


def find_second_worst_points_index(points):
    worst_points_index = find_worst_points_index
    return np.array([
        v['value'] for i, v in enumerate(points) if i != worst_points_index
    ]).argmax()


def find_best_points_index(points):
    return np.array([point['value'] for point in points]).argmin()


def shrink(f, points, gamma=0.5):
    X = []

    for i, x in enumerate(points):
        if i == find_best_points_index(points):
```

```
            X.append(x)
            continue

        best_point = points[find_best_points_index(points)]
        x_coords = best_point['coords'] + gamma * (x['coords'] -
                                                   best_point['coords'])
        X.append({"coords": x_coords, "value": f(x_coords)})

    return X


def nelder_mead(f, starting_point, tolerance=0.001):
    # Stat tracing
    triangles = []
    function_calls = 0

    # Generate Simplex Points
    simplex = generate_points(f, starting_point)
    function_calls += len(simplex)

    n = len(simplex) - 1  # Number of variables

    while True:
        # Select Worst Point
        worst_points_index = find_worst_points_index(simplex)

        # Find Centroid
        centroid = find_centroid(f, simplex, n, worst_points_index)
        function_calls += 1

        # Reflection
        xr = step(f, simplex, centroid, worst_points_index, 1)
        function_calls += 1

        triangles.append(copy.deepcopy(simplex))  # Stats

        # Try Expansion
        if xr['value'] <= simplex[find_best_points_index(
                simplex)]['value']:  # F(x_r) <= F(x^(0))
            xe = step(f, simplex, centroid, worst_points_index, 2)
            function_calls += 1

            if xe['value'] <= simplex[find_best_points_index(
                    simplex)]['value']:  # F(x_e) <= F(x^(0))
                simplex[worst_points_index] = xe
            else:
                simplex[worst_points_index] = xr
        # Reflected is fine
        elif xr['value'] <= simplex[find_second_worst_points_index(
```

```
                simplex)]['value']:
            simplex[worst_points_index] = xr
        # Inside contraction
        elif xr['value'] >= simplex[worst_points_index]['value']:
            xic = step(f, simplex, centroid, worst_points_index, -0.5)
            function_calls += 1

            if xic['value'] <= simplex[worst_points_index]['value']:
                simplex[worst_points_index] = xic
            # Shrink
            else:
                simplex = shrink(f, simplex)
                function_calls += n
        # Outside contraction
        else:
            xoc = step(f, simplex, centroid, worst_points_index, 0.5)
            function_calls += 1

            if xoc['value'] <= simplex[worst_points_index]['value']:
                simplex[worst_points_index] = xoc
            # Shrink
            else:
                simplex = shrink(f, simplex)
                function_calls += n

        if np.linalg.norm(simplex[find_worst_points_index(simplex)]['coords']
 -
                          simplex[find_best_points_index(simplex)]['coords']
                          ) <= tolerance:
            break

    return simplex, triangles, function_calls
```
Listing 3: Deformuojamo simplekso algoritmo realizacija su Python