Отчёт по лабораторной работе №1 по фундаментальным концепциям искусственного интеллекта на тему: “ Градиентный спуск и его модификации”

Выполнил студент группы М80-114СВ-24 Сипкин Владислав.

1. Функция Химмельблау:

а) Этот код включает:

Листинг 1 – Программа классического градиентного спуска функции Химмельблау

```
import time
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, diff

# функция Химмельблау
def f(x, y):
    return (x**2 + y - 11)**2 + (x + y**2 - 7)**2

# градиент функции Химмельблау
def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)

    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy

N = 100      # число итераций
xx = 5     # начальное значение x
yy = 5     # начальное значение y
lmd = 0.001   # шаг сходимости

x_plt = np.linspace(-5.0, 5.0, 100)     # 100 точек по оси x
y_plt = np.linspace(-5.0, 5.0, 100)     # 100 точек по оси y
x_plt, y_plt = np.meshgrid(x_plt, y_plt)    # создаём двумерную сетку
f_plt = f(x_plt, y_plt)     # вычисляем значения функции для каждой пары (x, y)

plt.ion()     # включение интерактивного режима отображения графиков
fig = plt.figure()      # создание окна
ax = fig.add_subplot(111, projection='3d')      # создание 3D осей

# подписи осей
```

```
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")

ax.plot_surface(x_plt, y_plt, f_plt, cmap='viridis')    # график функции
point = ax.scatter(xx, yy, f(xx, yy), color="red")      # отображение начальной
точки

E = 1e-6  # порог сходимости
trajectory_x, trajectory_y, trajectory_z = [xx], [yy], [f(xx, yy)]
for i in range(N):
    df_dx, df_dy = gradient(f, xx, yy)     # разбиение градиента функции,
переданного пользователем, на частные частные производные

    # проверка на сходимость
    grad_norm = np.linalg.norm([df_dx, df_dy])  # вычисление нормы градиента
    # Проверка на сходимость для новых значени, кроме начального
    if grad_norm < E and i != 0:
        print(f"Алгоритм сошелся на итерации {i}")
        break
    xx = xx - lmd * df_dx    # изменение координаты x точки на текущей иттерации
    yy = yy - lmd * df_dy    # изменение координаты y точки на текущей иттерации

    # вычисление положения траектории
    trajectory_x.append(xx)
    trajectory_y.append(yy)
    trajectory_z.append(f(xx, yy))

    # отображение нового положения точки
    point.remove()
    point = ax.scatter(xx, yy, f(xx, yy), color="red")
    ax.plot(trajectory_x, trajectory_y, trajectory_z, color="black")  # линия
траектории

    #перерисовка графика и задержка на 200 мс
    fig.canvas.draw()
    plt.pause(0.2)  # короткая пауза для анимации

plt.ioff()      # выключение интерактивного режима отображения графиков
print(f"Локальный минимум: x = {xx}, y = {yy}")
analytical_min = (3, 2)  # известный минимум
error = np.sqrt((xx - analytical_min[0])**2 + (yy - analytical_min[1])**2)
print(f"Для локального минимума {analytical_min} погрешность: {error}")
ax.scatter(xx, yy, f(xx, yy), color="blue")
plt.show()
```

- реализацию классического градиентного спуска для данной функции;
- реализацию метода символьного вычисления градиента от пользователя через simpy:

Листинг 2 – Функция градиента функции Химмельбау

```python
# градиент функции Химмельблау
def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного 
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)

    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy
```

Результат реализации классического градиентного спуска для данной функции включает:

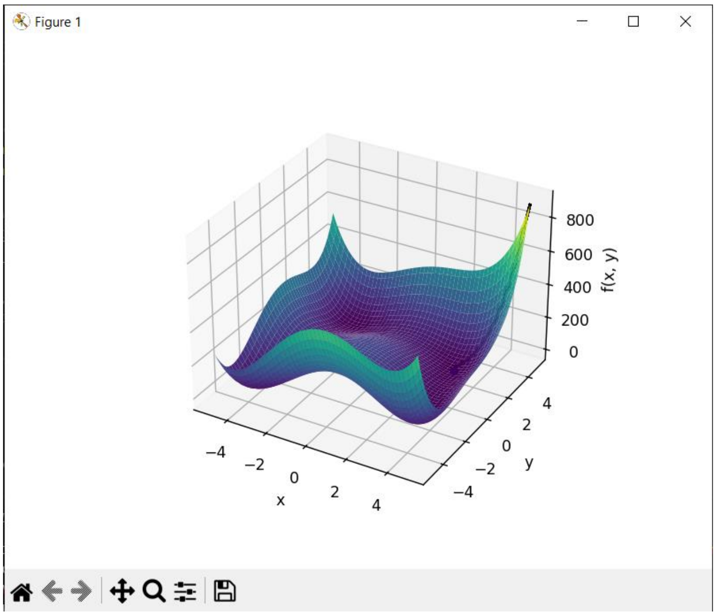


Рис. 1. Результат реализации классического градиентного спуска для функции Химмельблау

- визуализацию функции и точки оптимума (во время выполнения программы точка выражается красным цветом, а её траектория движения – чёрным цветом);
- вычисление погрешности найденного решения в сравнении с аналитическим для нескольких запусков:

```
Локальный минимум: x = 2.9822282470041976, y = 2.0413763094200457
Для локального минимума (3, 2) погрешность: 0.04503147994200468
```

Рис. 2. Результат поиска экспериментального локального минимума и сравнение с близким к нему локальным минимумом через подсчёт погрешности

- визуализацию точки найденного решения (точка выражена синим цветом после выполнения программы).

б) Код реализации моментной модификации градиентного спуска:

Листинг 3 – Программа градиентного спуска функции Химмельблау с моментной модификацией

```
import time
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, diff

# функция Химмельблау
def f(x, y):
    return (x**2 + y - 11)**2 + (x + y**2 - 7)**2

# градиент функции Химмельблау
def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)

    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy

N = 100      # число итераций
xx = 5     # начальное значение x
yy = 5     # начальное значение y
lmd = 0.001   # шаг сходимости
beta = 0.9      # коэффициент затухания
vx= 0     # начальное значение скорости x
vy= 0     # начальное значение скорости y
E = 1e-6  # порог сходимости
trajectory_x, trajectory_y, trajectory_z = [xx], [yy], [f(xx, yy)]      #
начальное положение траектории

x_plt = np.linspace(-5.0, 5.0, 100)     # 100 точек по оси x
y_plt = np.linspace(-5.0, 5.0, 100)     # 100 точек по оси y
x_plt, y_plt = np.meshgrid(x_plt, y_plt)    # создаём двумерную сетку
```

```
f_plt = f(x_plt, y_plt)     # вычисляем значения функции для каждой пары (x, y)

plt.ion()     # включение интерактивного режима отображения графиков
fig = plt.figure()      # создание окна
ax = fig.add_subplot(111, projection='3d')      # создание 3D осей

# подписи осей
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")

ax.plot_surface(x_plt, y_plt, f_plt, cmap='viridis')    # график функции
point = ax.scatter(xx, yy, f(xx, yy), color="red")      # отображение начальной 
точки

for i in range(N):
    df_dx, df_dy = gradient(f, xx, yy)     # разбиение градиента функции, 
переданного пользователем, на частные частные производные

    # проверка на сходимость
    grad_norm = np.linalg.norm([df_dx, df_dy])  # вычисление нормы градиента
    # Проверка на сходимость для новых значени, кроме начального
    if grad_norm < E and i != 0:
        print(f"Алгоритм сошелся на итерации {i}")
        break

    vx = beta * vx + (1 - beta) * df_dx    # изменение скорости точки по 
координате х на текущей иттерации
    vy = beta * vy + (1 - beta) * df_dy    # изменение скорости точки по 
координате н на текущей иттерации

    xx -= lmd * vx    # изменение координаты х точки на текущей иттерации
    yy -= lmd * vy    # изменение координаты у точки на текущей иттерации

    # vx = beta * vx + lmd * df_dx    # изменение скорости точки по координате х 
на текущей иттерации
    # vy = beta * vy + lmd * df_dy    # изменение скорости точки по координате н 
на текущей иттерации

    # xx -= vx    # изменение координаты х точки на текущей иттерации
    # yy -= vy    # изменение координаты у точки на текущей иттерации

    # вычисление положения траектории
    trajectory_x.append(xx)
    trajectory_y.append(yy)
    trajectory_z.append(f(xx, yy))

    # отображение нового положения точки
    point.remove()
    point = ax.scatter(xx, yy, f(xx, yy), color="red")
```

```
    ax.plot(trajectory_x, trajectory_y, trajectory_z, color="black")  # линия траектории

    #перерисовка графика и задержка на 200 мс
    fig.canvas.draw()
    plt.pause(0.2)  # короткая пауза для анимации

plt.ioff()      # выключение интерактивного режима отображения графиков
print(f"Локальный минимум: x = {xx}, y = {yy}")
analytical_min = (3, 2)  # известный минимум
error = np.sqrt((xx - analytical_min[0])**2 + (yy - analytical_min[1])**2)
print(f"Для локального минимума {analytical_min} погрешность: {error}")
ax.scatter(xx, yy, f(xx, yy), color="blue")
plt.show()
```

Результат реализации градиентного спуска с моментной модификацией для данной функции включает:

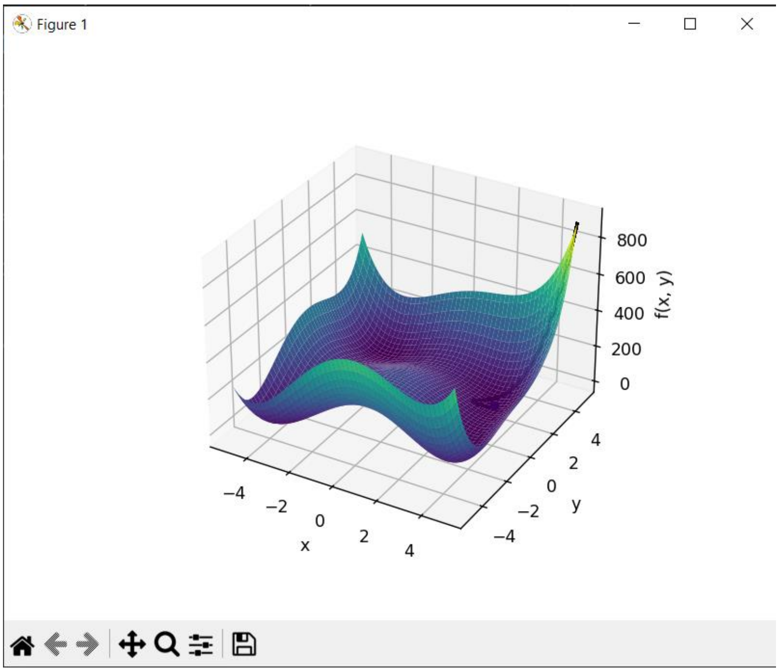


Рис. 3. Результат реализации градиентного спуска функции Химмельблау с моментной модификацией

```
Локальный минимум: x = 3.032090155077401, y = 1.9418602236627065
Для локального минимума (3, 2) погрешность: 0.06640791854471989
```

Рис. 4. Результат поиска экспериментального локального минимума и сравнение с близким к нему локальным минимумом через подсчёт погрешности

в) Код реализации адаптивной модификации градиентного спуска:

Листинг 4 – Программа градиентного спуска функции Химмельблау с адаптивной модификацией

```
import time
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, diff

# функция Химмельблау
def f(x, y):
    return (x**2 + y - 11)**2 + (x + y**2 - 7)**2

# градиент функции Химмельблау
def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)

    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy

N = 100      # число итераций
xx = 5     # начальное значение x
yy = 5     # начальное значение y
lmd = 0.9   # шаг сходимости
Gx = 0      # начальнаый квадрат градиента по dx
Gy = 0      # начальнаый квадрат градиента по dy
E = 1e-6  # порог сходимости
trajectory_x, trajectory_y, trajectory_z = [xx], [yy], [f(xx, yy)]      #
начальное положение траектории

x_plt = np.linspace(-5.0, 5.0, 100)     # 100 точек по оси x
y_plt = np.linspace(-5.0, 5.0, 100)     # 100 точек по оси y
x_plt, y_plt = np.meshgrid(x_plt, y_plt)    # создаём двумерную сетку
f_plt = f(x_plt, y_plt)     # вычисляем значения функции для каждой пары (x, y)

plt.ion()     # включение интерактивного режима отображения графиков
fig = plt.figure()      # создание окна
ax = fig.add_subplot(111, projection='3d')      # создание 3D осей

# подписи осей
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")
```

```
ax.plot_surface(x_plt, y_plt, f_plt, cmap='viridis')    # график функции
point = ax.scatter(xx, yy, f(xx, yy), color="red")      # отображение начальной
точки

for i in range(N):
    df_dx, df_dy = gradient(f, xx, yy)     # разбиение градиента функции,
переданного пользователем, на частные частные производные

    # проверка на сходимость
    grad_norm = np.linalg.norm([df_dx, df_dy])  # вычисление нормы градиента
    # Проверка на сходимость для новых значени, кроме начального
    if grad_norm < E and i != 0:
        print(f"Алгоритм сошелся на итерации {i}")
        break

    Gx += df_dx**2    # накопление квадрата градиента по dx
    Gy += df_dy**2    # накопление квадрата градиента по dн

    xx -= (lmd / np.sqrt(Gx + E)) * df_dx    # изменение координаты x точки на
текущей иттерации
    yy -= (lmd / np.sqrt(Gy + E)) * df_dy    # изменение координаты y точки на
текущей иттерации

    # вычисление положения траектории
    trajectory_x.append(xx)
    trajectory_y.append(yy)
    trajectory_z.append(f(xx, yy))

    # отображение нового положения точки
    point.remove()
    point = ax.scatter(xx, yy, f(xx, yy), color="red")
    ax.plot(trajectory_x, trajectory_y, trajectory_z, color="black")  # линия
траектории

    #перерисовка графика и задержка на 200 мс
    fig.canvas.draw()
    plt.pause(0.2)  # короткая пауза для анимации

plt.ioff()      # выключение интерактивного режима отображения графиков
print(f"Локальный минимум: x = {xx}, y = {yy}")
analytical_min = (3, 2)  # известный минимум
error = np.sqrt((xx - analytical_min[0])**2 + (yy - analytical_min[1])**2)
print(f"Для локального минимума {analytical_min} погрешность: {error}")
ax.scatter(xx, yy, f(xx, yy), color="blue")
plt.show()
```

Результат реализации градиентного спуска с адаптивной модификацией для данной функции включает:

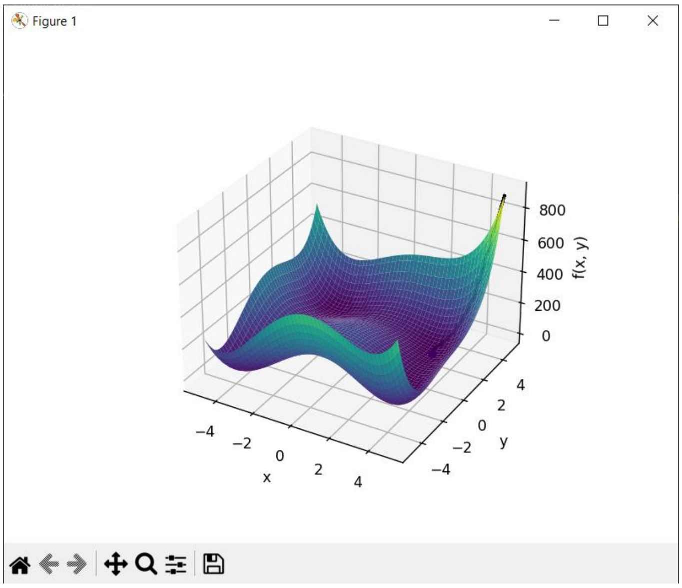


Рис. 5. Результат реализации градиентного спуска функции Химмельблау с адаптивной модификацией

```
Локальный минимум: x = 2.995288503560542, y = 2.012311485362971
Для локального минимума (3, 2) погрешность: 0.013182217967461935
```

Рис. 6. Результат поиска экспериментального локального минимума и сравнение с близким к нему локальным минимумом через подсчёт погрешности

г) Код реализации метода эволюции темпа обучения, использовав экспоненциальную зависимость:

Листинг 5 – Программа градиентного спуска функции Химмельблау с методом эволюции темпа обучения, использовав экспоненциальную зависимость:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, diff

# функция Химмельблау
def f(x, y):
    return (x**2 + y - 11)**2 + (x + y**2 - 7)**2

# градиент функции Химмельблау
def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)
```

```
    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy

N = 100      # число итераций
xx = 5     # начальное значение x
yy = 5     # начальное значение y
lmd = 0.01   # уначальный шаг сходимости
lambda_ = 0.05

x_plt = np.linspace(-5.0, 5.0, 100)     # 100 точек по оси x
y_plt = np.linspace(-5.0, 5.0, 100)     # 100 точек по оси y
x_plt, y_plt = np.meshgrid(x_plt, y_plt)    # создаём двумерную сетку
f_plt = f(x_plt, y_plt)     # вычисляем значения функции для каждой пары (x, y)

plt.ion()     # включение интерактивного режима отображения графиков
fig = plt.figure()      # создание окна
ax = fig.add_subplot(111, projection='3d')      # создание 3D осей

# подписи осей
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")

ax.plot_surface(x_plt, y_plt, f_plt, cmap='viridis')    # график функции
point = ax.scatter(xx, yy, f(xx, yy), color="red")      # отображение начальной
точки

E = 1e-6  # порог сходимости
trajectory_x, trajectory_y, trajectory_z = [xx], [yy], [f(xx, yy)]
for i in range(N):
    df_dx, df_dy = gradient(f, xx, yy)     # разбиение градиента функции,
переданного пользователем, на частные частные производные

    # проверка на сходимость
    grad_norm = np.linalg.norm([df_dx, df_dy])  # вычисление нормы градиента
    # Проверка на сходимость для новых значени, кроме начального
    if grad_norm < E and i != 0:
        print(f"Алгоритм сошелся на итерации {i}")
        break

    lmd = lmd * np.exp(-lambda_ * i)    # экспоненциальное уменьшение шага
обучения

    xx = xx - lmd * df_dx    # изменение координаты x точки на текущей иттерации
    yy = yy - lmd * df_dy    # изменение координаты y точки на текущей иттерации

    # вычисление положения траектории
    trajectory_x.append(xx)
```

```
    trajectory_y.append(yy)
    trajectory_z.append(f(xx, yy))

    # отображение нового положения точки
    point.remove()
    point = ax.scatter(xx, yy, f(xx, yy), color="red")
    ax.plot(trajectory_x, trajectory_y, trajectory_z, color="black")  # линия траектории

    #перерисовка графика и задержка на 200 мс
    fig.canvas.draw()
    plt.pause(0.2)  # короткая пауза для анимации

plt.ioff()      # выключение интерактивного режима отображения графиков
print(f"Локальный минимум: x = {xx}, y = {yy}")
analytical_min = (3, 2)  # известный минимум
error = np.sqrt((xx - analytical_min[0])**2 + (yy - analytical_min[1])**2)
print(f"Для локального минимума {analytical_min} погрешность: {error}")
ax.scatter(xx, yy, f(xx, yy), color="blue")
plt.show()
```

Результат реализации градиентного спуска с методом эволюции темпа обучения данной функции, использовав экспоненциальную зависимость:

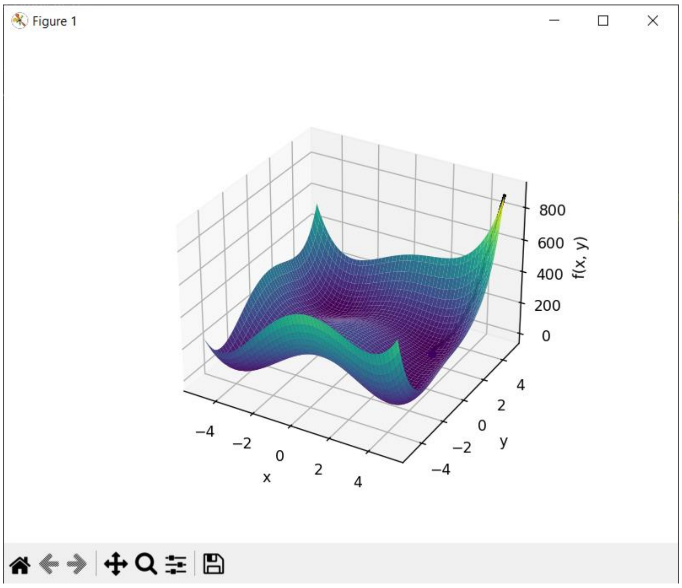


Рис. 7. Результат реализации градиентного спуска функции Химмельбау с методом эволюции темпа обучения, использовав экспоненциальную зависимость

```
Локальный минимум: x = 2.995288503560542, y = 2.012311485362971
Для локального минимума (3, 2) погрешность: 0.013182217967461935
```

Рис. 8. Результат поиска экспериментального локального минимума и сравнение с близким к нему локальным минимумом через подсчёт погрешности

1. Функция трёхгорбового верблюда:

а) Этот код включает:

Листинг 6 – Программа классического градиентного спуска функции трёхгорбового верблюда

```
import time
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, diff

# функция трехгорбого верблюда
def f(x, y):
    return 0.26 * (x**2 + y**2) - 0.48 * x * y

# градиент функции трехгорбого верблюда

def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)

    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy

N = 50      # число итераций
xx = 3     # начальное значение x
yy = -6     # начальное значение y
lmd = 0.9   # шаг сходимости

x_plt = np.linspace(-10.0, 10.0, 200)     # 200 точек по оси x
y_plt = np.linspace(-10.0, 10.0, 200)      # 200 точек по оси y
x_plt, y_plt = np.meshgrid(x_plt, y_plt)    # создаём двумерную сетку
f_plt = f(x_plt, y_plt)     # вычисляем значения функции для каждой пары (x, y)

plt.ion()     # включение интерактивного режима отображения графиков
fig = plt.figure()      # создание окна
ax = fig.add_subplot(111, projection='3d')      # создание 3D осей

# подписи осей
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")
```

```
ax.plot_surface(x_plt, y_plt, f_plt, cmap='viridis')    # график функции
point = ax.scatter(xx, yy, f(xx, yy), color="red")      # отображение начальной
точки

E = 1e-6  # порог сходимости
trajectory_x, trajectory_y, trajectory_z = [xx], [yy], [f(xx, yy)]
for i in range(N):
    df_dx, df_dy = gradient(f, xx, yy)     # разбиение градиента функции,
переданного пользователем, на частные частные производные

    # проверка на сходимость
    grad_norm = np.linalg.norm([df_dx, df_dy])  # вычисление нормы градиента
    # Проверка на сходимость для новых значени, кроме начального
    if grad_norm < E and i != 0:
        print(f"Алгоритм сошелся на итерации {i}")
        break
    xx = xx - lmd * df_dx    # изменение координаты x точки на текущей иттерации
    yy = yy - lmd * df_dy    # изменение координаты y точки на текущей иттерации

    # вычисление положения траектории
    trajectory_x.append(xx)
    trajectory_y.append(yy)
    trajectory_z.append(f(xx, yy))

    # отображение нового положения точки
    point.remove()
    point = ax.scatter(xx, yy, f(xx, yy), color="red")
    ax.plot(trajectory_x, trajectory_y, trajectory_z, color="black")  # линия
траектории

    #перерисовка графика и задержка на 200 мс
    fig.canvas.draw()
    plt.pause(0.2)  # короткая пауза для анимации

plt.ioff()      # выключение интерактивного режима отображения графиков
print(f"Локальный минимум: x = {xx}, y = {yy}")
analytical_min = (0, 0)  # известный минимум
error = np.sqrt((xx - analytical_min[0])**2 + (yy - analytical_min[1])**2)
print(f"Для локального минимума {analytical_min} погрешность: {error}")
ax.scatter(xx, yy, f(xx, yy), color="blue")
plt.show()
```

- реализацию классического градиентного спуска для данной функции;
- реализацию метода символьного вычисления градиента от пользователя через simpy:

Листинг 7 – Функция градиента функции трёхгорбового верблюда

```
# градиент функции трёхгорбового верблюда
```

```
def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)

    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy
```

Результат реализации классического градиентного спуска для данной функции включает:

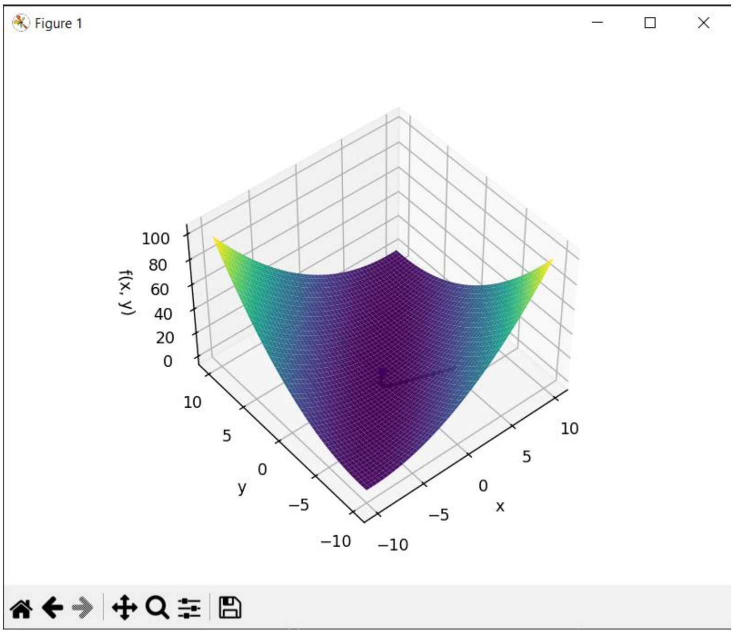


Рис. 9. Результат реализации классического градиентного спуска для функции трёхгорбового верблюда

- визуализацию функции и точки оптимума (во время выполнения программы точка выражается красным цветом, а её траектория движения – чёрным цветом);
- вычисление погрешности найденного решения в сравнении с аналитическим для нескольких запусков:

```
Локальный минимум: x = -0.23985178462559206, y = -0.23985178462559206
Для локального минимума (0, 0) погрешность: 0.3392016467769029
```

Рис. 10. Результат поиска экспериментального локального минимума и сравнение с близким к нему локальным минимумом через подсчёт погрешности

- визуализацию точки найденного решения (точка выражена синим цветом после выполнения программы).

б) Код реализации моментной модификации градиентного спуска:

Листинг 8 – Программа градиентного спуска функции трёхгорбового верблюда с моментной модификацией

```
import time
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, diff

# функция трехгорбого верблюда
def f(x, y):
    return 0.26 * (x**2 + y**2) - 0.48 * x * y

# градиент функции трехгорбого верблюда

def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)

    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy

N = 50      # число итераций
xx = 3     # начальное значение x
yy = -6     # начальное значение y
lmd = 0.9   # шаг сходимости
beta = 0.9      # коэффициент затухания
vx= 0     # начальное значение скорости x
vy= 0     # начальное значение скорости y
E = 1e-6  # порог сходимости
trajectory_x, trajectory_y, trajectory_z = [xx], [yy], [f(xx, yy)]      #
начальное положение траектории

x_plt = np.linspace(-10.0, 10.0, 200)     # 200 точек по оси x
y_plt = np.linspace(-10.0, 10.0, 200)      # 200 точек по оси y
x_plt, y_plt = np.meshgrid(x_plt, y_plt)    # создаём двумерную сетку
f_plt = f(x_plt, y_plt)     # вычисляем значения функции для каждой пары (x, y)

plt.ion()     # включение интерактивного режима отображения графиков
```

```
fig = plt.figure()      # создание окна
ax = fig.add_subplot(111, projection='3d')      # создание 3D осей

# подписи осей
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")

ax.plot_surface(x_plt, y_plt, f_plt, cmap='viridis')    # график функции
point = ax.scatter(xx, yy, f(xx, yy), color="red")      # отображение начальной
точки

for i in range(N):
    df_dx, df_dy = gradient(f, xx, yy)      # разбиение градиента функции,
переданного пользователем, на частные частные производные

    # проверка на сходимость
    grad_norm = np.linalg.norm([df_dx, df_dy])  # вычисление нормы градиента
    # Проверка на сходимость для новых значени, кроме начального
    if grad_norm < E and i != 0:
        print(f"Алгоритм сошелся на итерации {i}")
        break

    vx = beta * vx + (1 - beta) * df_dx    # изменение скорости точки по
координате x на текущей иттерации
    vy = beta * vy + (1 - beta) * df_dy    # изменение скорости точки по
координате н на текущей иттерации

    xx -= lmd * vx    # изменение координаты x точки на текущей иттерации
    yy -= lmd * vy    # изменение координаты y точки на текущей иттерации

    # vx = beta * vx + lmd * df_dx    # изменение скорости точки по координате x
на текущей иттерации
    # vy = beta * vy + lmd * df_dy    # изменение скорости точки по координате н
на текущей иттерации

    # xx -= vx    # изменение координаты x точки на текущей иттерации
    # yy -= vy    # изменение координаты y точки на текущей иттерации

    # вычисление положения траектории
    trajectory_x.append(xx)
    trajectory_y.append(yy)
    trajectory_z.append(f(xx, yy))

    # отображение нового положения точки
    point.remove()
    point = ax.scatter(xx, yy, f(xx, yy), color="red")
    ax.plot(trajectory_x, trajectory_y, trajectory_z, color="black")  # линия
траектории

    #перерисовка графика и задержка на 200 мс
```

```
    fig.canvas.draw()
    plt.pause(0.2)  # короткая пауза для анимации

plt.ioff()      # выключение интерактивного режима отображения графиков
print(f"Локальный минимум: x = {xx}, y = {yy}")
analytical_min = (3, 2)  # известный минимум
error = np.sqrt((xx - analytical_min[0])**2 + (yy - analytical_min[1])**2)
print(f"Для локального минимума {analytical_min} погрешность: {error}")
ax.scatter(xx, yy, f(xx, yy), color="blue")
plt.show()
```

Результат реализации градиентного спуска с моментной модификацией для данной функции включает:

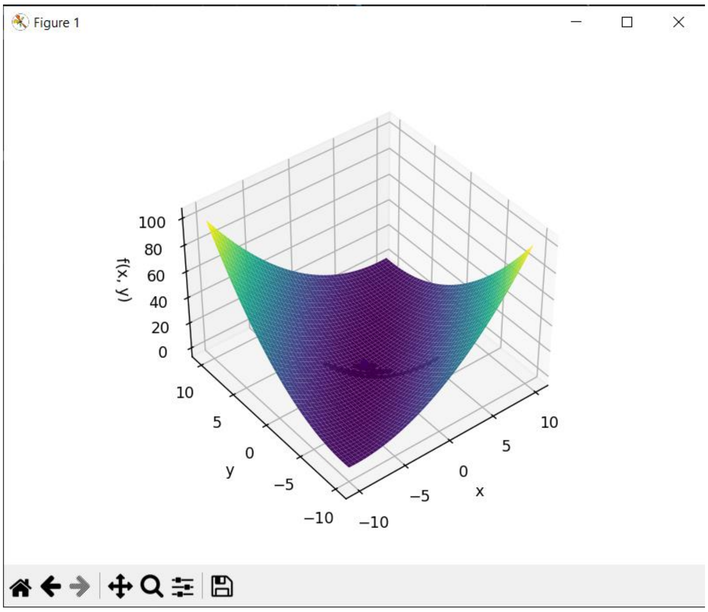


Рис. 11. Результат реализации градиентного спуска функции трёхгорбового верблюда с моментной модификацией

```
Локальный минимум: x = -0.4532270804573275, y = 0.1159286058357167
Для локального минимума (0, 0) погрешность: 0.4678185846146835
```

Рис. 12. Результат поиска экспериментального локального минимума и сравнение с близким к нему локальным минимумом через подсчёт погрешности

в) Код реализации адаптивной модификации градиентного спуска:

Листинг 9 – Программа градиентного спуска функции трёхгорбового верблюда с адаптивной модификацией

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

```python
from sympy import symbols, diff

# функция трехгорбого верблюда
def f(x, y):
    return 0.26 * (x**2 + y**2) - 0.48 * x * y

# градиент функции трехгорбого верблюда

def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)

    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy

N = 100      # число итераций
xx = 4     # начальное значение x
yy = -8     # начальное значение y
lmd = 0.9   # шаг сходимости
Gx = 0      # начальнаый квадрат градиента по dx
Gy = 0      # начальнаый квадрат градиента по dy
E = 1e-6  # порог сходимости
trajectory_x, trajectory_y, trajectory_z = [xx], [yy], [f(xx, yy)]      #
начальное положение траектории

x_plt = np.linspace(-10.0, 10.0, 200)     # 200 точек по оси x
y_plt = np.linspace(-10.0, 10.0, 200)      # 200 точек по оси y
x_plt, y_plt = np.meshgrid(x_plt, y_plt)    # создаём двумерную сетку
f_plt = f(x_plt, y_plt)     # вычисляем значения функции для каждой пары (x, y)

plt.ion()     # включение интерактивного режима отображения графиков
fig = plt.figure()      # создание окна
ax = fig.add_subplot(111, projection='3d')      # создание 3D осей

# подписи осей
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")

ax.plot_surface(x_plt, y_plt, f_plt, cmap='viridis')    # график функции
point = ax.scatter(xx, yy, f(xx, yy), color="red")      # отображение начальной
точки

for i in range(N):
```

```
    df_dx, df_dy = gradient(f, xx, yy)     # разбиение градиента функции, переданного пользователем, на частные частные производные

    # проверка на сходимость
    grad_norm = np.linalg.norm([df_dx, df_dy])  # вычисление нормы градиента
    # Проверка на сходимость для новых значени, кроме начального
    if grad_norm < E and i != 0:
        print(f"Алгоритм сошелся на итерации {i}")
        break

    Gx += df_dx**2    # накопление квадрата градиента по dx
    Gy += df_dy**2    # накопление квадрата градиента по dн

    xx -= (lmd / np.sqrt(Gx + E)) * df_dx    # изменение координаты x точки на текущей иттерации
    yy -= (lmd / np.sqrt(Gy + E)) * df_dy    # изменение координаты y точки на текущей иттерации

    # вычисление положения траектории
    trajectory_x.append(xx)
    trajectory_y.append(yy)
    trajectory_z.append(f(xx, yy))

    # отображение нового положения точки
    point.remove()
    point = ax.scatter(xx, yy, f(xx, yy), color="red")
    ax.plot(trajectory_x, trajectory_y, trajectory_z, color="black")  # линия траектории

    #перерисовка графика и задержка на 200 мс
    fig.canvas.draw()
    plt.pause(0.2)  # короткая пауза для анимации

plt.ioff()      # выключение интерактивного режима отображения графиков
print(f"Локальный минимум: x = {xx}, y = {yy}")
analytical_min = (0, 0)  # известный минимум
error = np.sqrt((xx - analytical_min[0])**2 + (yy - analytical_min[1])**2)
print(f"Для локального минимума {analytical_min} погрешность: {error}")
ax.scatter(xx, yy, f(xx, yy), color="blue")
plt.show()
```

Результат реализации градиентного спуска с адаптивной модификацией для данной функции включает:

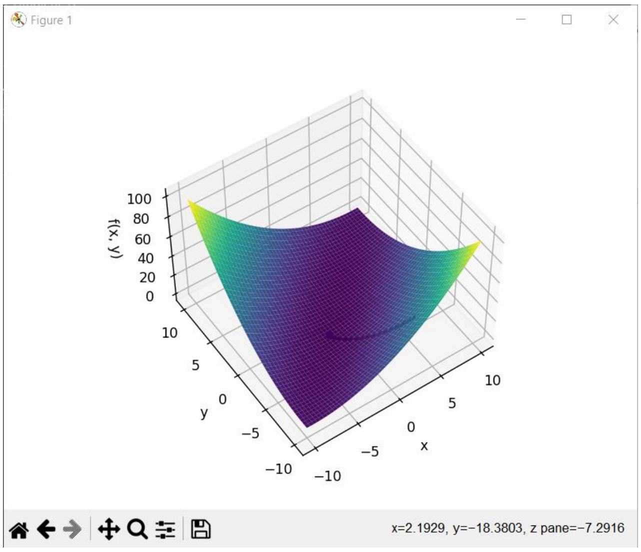


Рис. 13. Результат реализации градиентного спуска функции трёхгорбового верблюда с адаптивной модификацией

```
Локальный минимум: x = -1.6082491915453812, y = -1.6228819952057363
Для локального минимума (0, 0) погрешность: 2.284778202029537
```

Рис. 14. Результат поиска экспериментального локального минимума и сравнение с близким к нему локальным минимумом через подсчёт погрешности

г) Код реализации метода эволюции темпа обучения, использовав экспоненциальную зависимость:

Листинг 10 – Программа градиентного спуска функции трёхгорбового верблюда с методом эволюции темпа обучения, использовав экспоненциальную зависимость:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, diff

# функция трехгорбого верблюда
def f(x, y):
    return 0.26 * (x**2 + y**2) - 0.48 * x * y

# градиент функции трехгорбого верблюда
def gradient(func, x, y):
    x_sym, y_sym = symbols('x y') # определение переменных для символьного
дифференцирования
    f_sym = func(x_sym, y_sym) # символьное представление функции

    # символьные частные производные
    df_dx_sym = diff(f_sym, x_sym)
    df_dy_sym = diff(f_sym, y_sym)
```

```
    # Подставляем значения x и y в символические производные
    df_dx = float(df_dx_sym.subs({x_sym: x, y_sym: y}))
    df_dy = float(df_dy_sym.subs({x_sym: x, y_sym: y}))
    return df_dx, df_dy

N = 50      # число итераций
xx = 3     # начальное значение x
yy = -6     # начальное значение y
lmd = 0.9   # шаг сходимости
lambda_ = 0.05

x_plt = np.linspace(-10.0, 10.0, 200)     # 200 точек по оси x
y_plt = np.linspace(-10.0, 10.0, 200)      # 200 точек по оси y
x_plt, y_plt = np.meshgrid(x_plt, y_plt)    # создаём двумерную сетку
f_plt = f(x_plt, y_plt)     # вычисляем значения функции для каждой пары (x, y)

plt.ion()     # включение интерактивного режима отображения графиков
fig = plt.figure()      # создание окна
ax = fig.add_subplot(111, projection='3d')      # создание 3D осей

# подписи осей
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")

ax.plot_surface(x_plt, y_plt, f_plt, cmap='viridis')    # график функции
point = ax.scatter(xx, yy, f(xx, yy), color="red")      # отображение начальной
точки

E = 1e-6  # порог сходимости
trajectory_x, trajectory_y, trajectory_z = [xx], [yy], [f(xx, yy)]
for i in range(N):
    df_dx, df_dy = gradient(f, xx, yy)     # разбиение градиента функции,
переданного пользователем, на частные частные производные

    # проверка на сходимость
    grad_norm = np.linalg.norm([df_dx, df_dy])  # вычисление нормы градиента
    # Проверка на сходимость для новых значени, кроме начального
    if grad_norm < E and i != 0:
        print(f"Алгоритм сошелся на итерации {i}")
        break

    lmd = lmd * np.exp(-lambda_ * i)    # экспоненциальное уменьшение шага
обучения

    xx = xx - lmd * df_dx    # изменение координаты x точки на текущей иттерации
    yy = yy - lmd * df_dy    # изменение координаты y точки на текущей иттерации

    # вычисление положения траектории
    trajectory_x.append(xx)
    trajectory_y.append(yy)
```

```
    trajectory_z.append(f(xx, yy))

    # отображение нового положения точки
    point.remove()
    point = ax.scatter(xx, yy, f(xx, yy), color="red")
    ax.plot(trajectory_x, trajectory_y, trajectory_z, color="black")  # линия траектории

    #перерисовка графика и задержка на 200 мс
    fig.canvas.draw()
    plt.pause(0.2)  # короткая пауза для анимации

plt.ioff()      # выключение интерактивного режима отображения графиков
print(f"Локальный минимум: x = {xx}, y = {yy}")
analytical_min = (0, 0)  # известный минимум
error = np.sqrt((xx - analytical_min[0])**2 + (yy - analytical_min[1])**2)
print(f"Для локального минимума {analytical_min} погрешность: {error}")
ax.scatter(xx, yy, f(xx, yy), color="blue")
plt.show()
```

Результат реализации градиентного спуска с методом эволюции темпа обучения данной функции, использовав экспоненциальную зависимость:

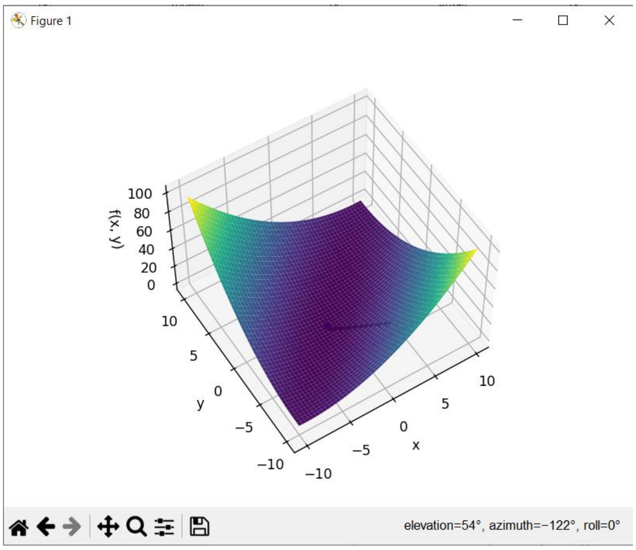


Рис. 15. Результат реализации градиентного спуска функции трёхгорбового верблюда с методом эволюции темпа обучения, использовав экспоненциальную зависимость

```
Локальный минимум: x = -1.220660556616891, y = -1.2215765687057751
Для локального минимума (0, 0) погрешность: 1.7269225540513489
```

Рис. 16. Результат поиска экспериментального локального минимума и сравнение с близким к нему локальным минимумом через подсчёт погрешности