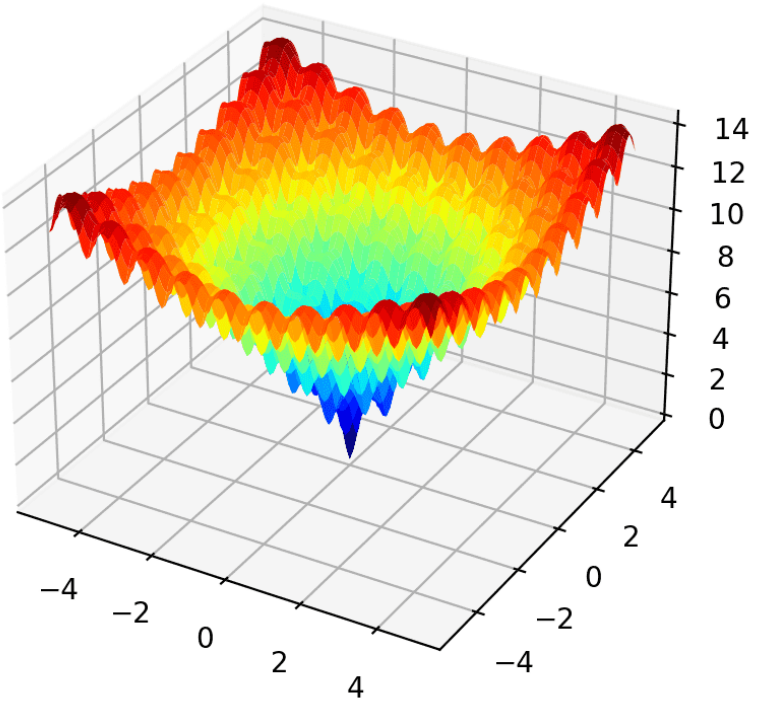
Міністерство освіти і науки України
Національний технічний університет України
"Київський політехнічний інститут імені Ігоря Сікорського"
Фізико-технічний інститут

ЛАБОРАТОРНА РОБОТА
з дисципліни: “ Методи оптимізації ”
Тема:“ Метод Ньютона,
Градієнтний спуск з подрібненням кроку
Метод найшвидшого спуску
Метод спряжених градієнтів ClonAlg ”

Виконала бригада №4:
студенти гр. ФІ-94, Кріпака Ілля, Куценко Андрій
та студент гр.ФІ-91, Хохлов Ярослав

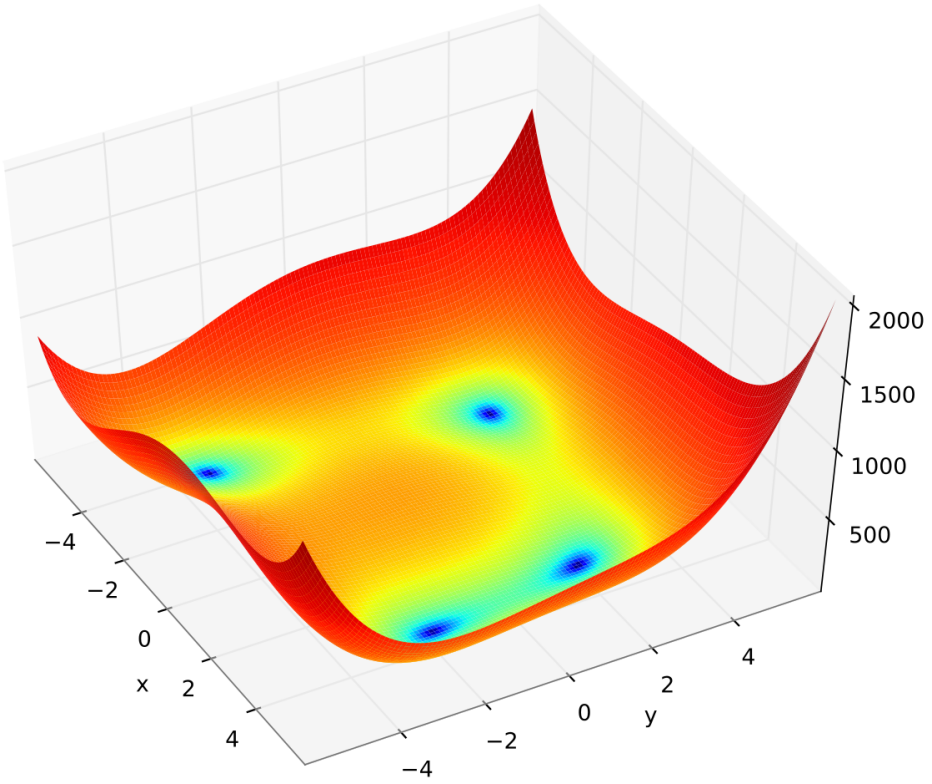
Досліджувані функції

Функція Еклі



$$f(x,y) = -20\exp\left[-0.2\sqrt{0.5\left(x^2+y^2\right)}\right] - \exp[0.5\left(\cos 2\pi x + \cos 2\pi y\right)] + e + 20$$

Функція Химмельблау



$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2.$$

Метод Ньютона(Метод дотичних)

Метод Ньютона або **Метод дотичних** - це ітераційний чисельний метод знаходження нуля заданої функції.

Суть:
Для того, щоб вирішити рівняння $f(x) = 0$ методом простої ітерації, його треба звести до еквівалентного рівняння $x = \varphi(x)$ де φ - стискуюче відображення.

Основна ідея методу полягає в наступному:

- задається початкове наближення поблизу ймовірного кореня
- після чого будується дотична до графіка досліджуваної функції в точці наближення, для якої перетин з віссю абсцис.
- Ця точка береться як наступне наближення.
- І так далі, поки не буде досягнуто необхідної точності.

Алгоритм:

1) Задається початкове наближення x_0
2) Поки не виконана умова зупинки, якою можна взяти $|x_{n+1} - x_n| < \varepsilon$ (тобто є похибка в потрібному інтервалі)
3) Обчилслюють нове наближення $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$

```
## NEWTON METHOD
 def newton(self, X, direction, threshold=None, iters_num=100):
        f = lambda p: self.function(X + p * direction)
        x = 0
        x_new = x

        if (threshold and iters_num) or (not threshold and not
iters_num):
            raise Exception("You must specify either number of
iterations or threshold.")

        if not threshold:
            for i in range(iters_num):
                der1 = self.derivative(f, x)
                der2 = self.derivative(f, x, order=2)
                assert (der2 != 0)

                x_new = x - der1 / der2

        elif not iters_num:
            der1 = self.derivative(f, x)
            while abs(der1) > threshold:
                der2 = self.derivative(f, x, order=2)
                assert (der2 != 0)

                x_new = x - der1 / der2
                der1 = self.derivative(f, x)

        return x_new
```

Випадковий пошук

Випадковий пошук - метод випадкового пошуку, званий також **методом Монте-Карло**

(**Метод Монте-Карло** — це метод імітації для приблизного відтворення реальних явищ. Він об'єднує аналіз чутливості (сприйнятливості) і аналіз розподілу ймовірностей вхідних змінних. Цей метод дає змогу побудувати модель, мінімізуючи дані, а також максимізувати значення даних, які використовуються в моделі.)

заснований на тому, що при тому самому числі випробувань ймовірність отримання рішення, близького до оптимального, при випадковому пошуку

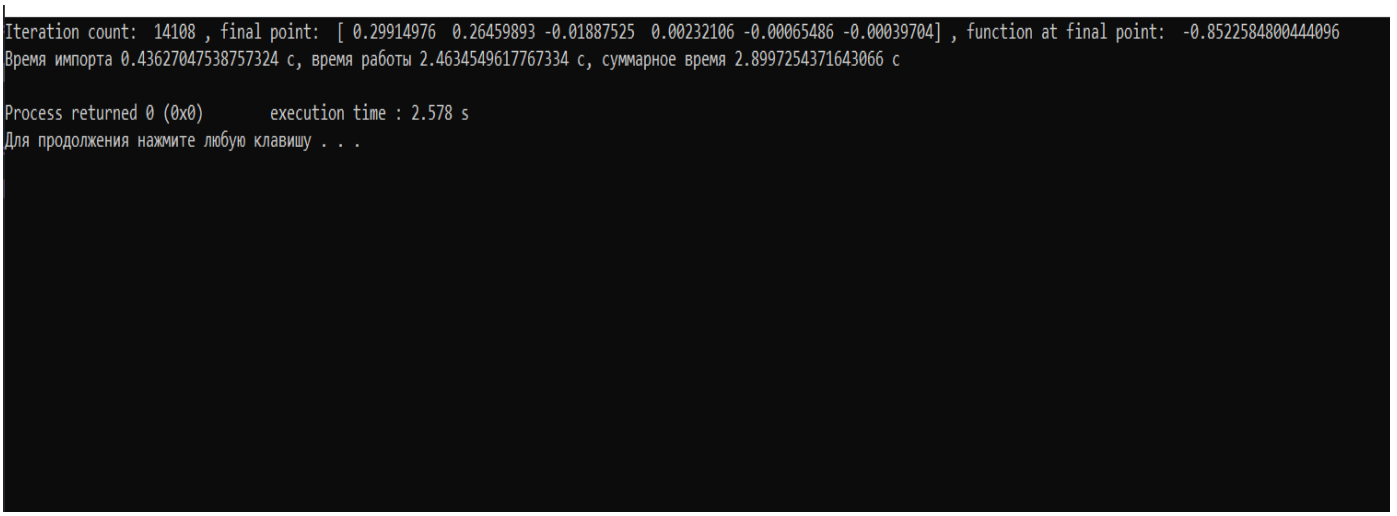
більше, ніж при послідовному переборі через рівні інтервали зміни окремих параметрів.

Один із найпростіших алгоритмів - локальний неадаптивний алгоритм випадкового пошуку
Алгоритм:

1) Задаємо початкову точку, представлену вектором X0, оголошуємо її поточною та обчислюємо у ній значень цільової функції.
2) Поточній точці надаємо збільшення у вигляді випадкового вектора дельта X і обчислюється значення цільової функції.
3) Якщо значення цільової функції покращилося, то цю точку робимо поточною.
4) Перевірити умову зупинки. Якщо воно виконується, то переходимо на крок 5, інакше на крок 2.
5) Зупини.

Виходячи із алгоритму що вище, він не вимагає обчислення похідних цільової функції визначальним її змінним, що є +.

Перевагами даного алгоритму є його простота, стійкість та інтуїтивна зрозумілість, не дивлячись на об’єм обчислень.
Недоліками – низька швидкість збіжності, а також невизначеність у виборі умови зупинки.



```
## RANDOM SEARCH
import time
start_time = time.time()

from numpy import pi,cos,sin,abs,sqrt,exp,array,prod,sum
from numpy.random import uniform

t1=time.time() - start_time

n=6
def f(x):
    x=array(x).T
    return -exp(sum([-x[i]**2 for i in range(0,n)]))
    # cos(x[0])*sin(x[1])*sin(x[2])*sin(x[3])*sin(x[4])

x=uniform(-1, 1,n)

d=0.01
eps1=0.0001

i=0
while d>eps1:
    i+=1
    if i>10**5:
        print('Exception: iteration overflow!')
        break
    xo = x
    psi,psi[0],psi[1]=uniform(0, pi,n),0,uniform(0, 2*pi)
    h=d*array([cos(psi[i-1])*prod([sin(psi[k]) for k in range(i,n)]) for
i in range(1,n+1)])
    # А.Ф. Никифоров, С.К.Суслов, В.Б.Уваров. Классические
ортогональные полиномы дискретной переменной.
    # М.:Наука, 1985, 161-я страница.

    x=x+h
    if f(x)>f(xo):
        d=d-0.01*d
        x=xo
    if f(x)<f(xo):
        d=d+0.01*d

t2=time.time() - start_time
print('Iteration count: ',i, ', final point: ',x,', function at final
point: ',f(x))
print("Время импорта %s с, время работы %s с, суммарное время %s с" %
(t1,t2,t1+t2))
```

Алгоритм Нелдера Міда

(**метод недеформованого багатогранника**)

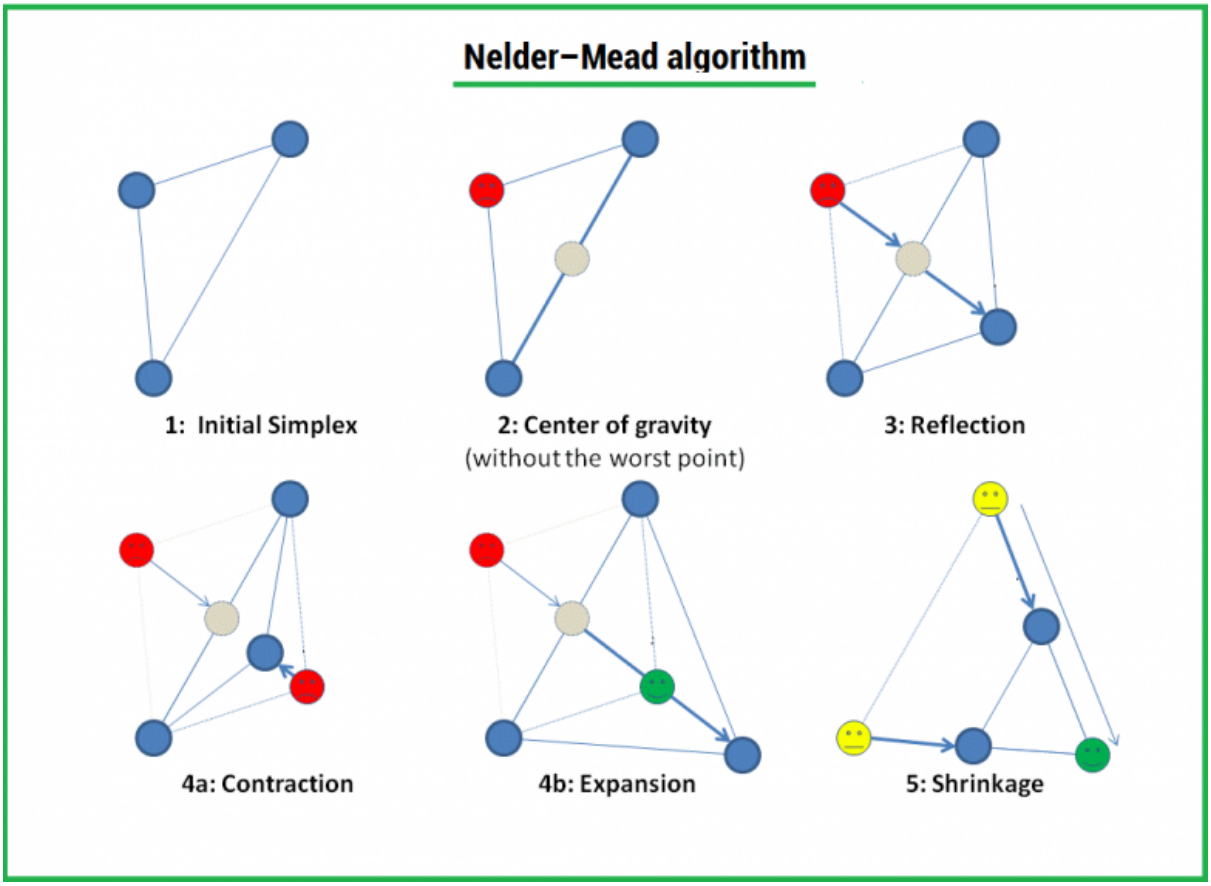
Алгоритм Нелдера Міда - метод безумовної оптимізації функції від кількох змінних, який не використовує похідної (градієнтів) функції, а тому легко застосовується до негладких та/або зашумлених функцій що полягає у формуванні **симплексу** (simplex) та подальшого його деформування у напрямку мінімуму, за допомогою трьох операцій:

1) Відображення (reflection);
2) Розтягнення (expansion);
3) Стиснення (contract);

Симплекс є геометричною фігурою, що є n — мірним узагальненням трикутника. Для одномірного простору це відрізок, для двовимірного трикутник. Таким чином, n - вимірний симплекс має n + 1 вершину.

Метод знаходить локальний екстремум і може застрягти в одному з них. Якщо все ж таки потрібно знайти глобальний екстремум, можна намагатися вибирати інший початковий симплекс.

Алгоритм закінчується, коли:
1) Було виконано необхідну кількість ітерацій.
2) Площа симплексу досягає певної величини.
3) Поточне найкраще рішення досягло необхідної точності.




```
import numpy as np
from numpy import
sqrt,sum,hstack,ones,zeros,array,fill_diagonal,argsort,sin,cos
import matplotlib.animation as animation
import matplotlib.patches as patches
import matplotlib.pyplot as plt
# Вычислим минимум функции Розенброка. Известно, что он в точке [1,1]
# Вычислим минимум функции sin(x)*cos(y). известно, что один из
минимумов в точке [-pi/2,0]
# Хитрое объявление, чтобы работать и с одним вектором [x1, y1], и
набором векторов [[x1,y1],[x2,y2]]
def f(x):
      x=array(x).T
      return (sin(x[0])*cos(x[1])).T
      # return (100*(x[1]-x[0]**2)**2+(1-x[0])**2).T

# Размерность пространства, расстояние между вершинами начального
симплекса, условие сходимости
n,t,eps=2,1,0.000001
# Параметры отражения, сжатия, растяжения
alpha,beta,gamma=1,0.5,2

# Построение регулярного симплекса
```

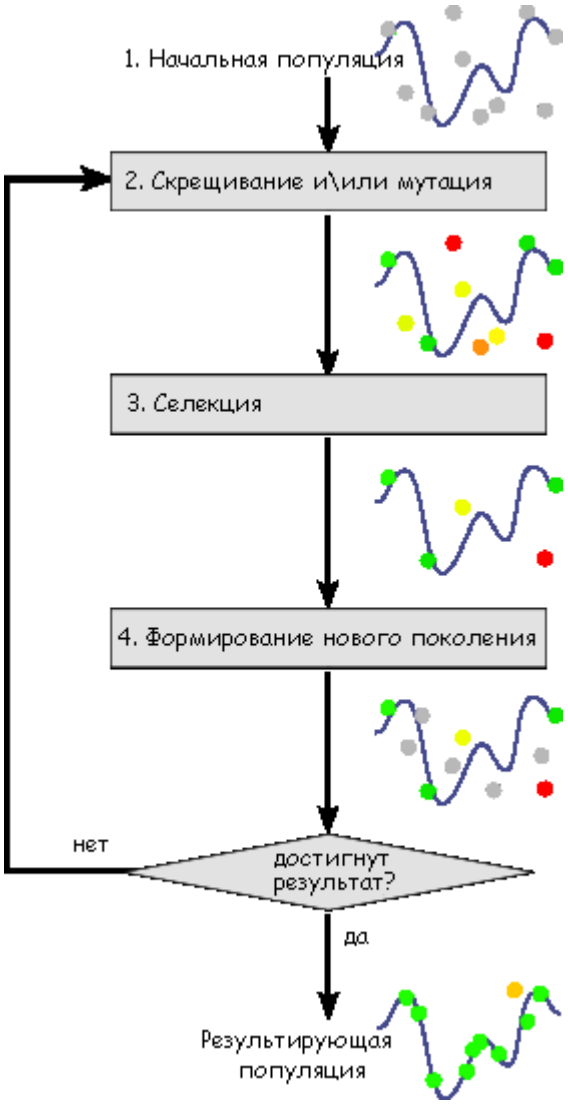
```
d1=(sqrt(n+1)+n-1)*t/sqrt(2)/n
d2=(d1-1/sqrt(2))*t
nn=ones([n,n])*d2
fill_diagonal(nn, d1)
simplex=hstack([zeros([n,1]),nn]).T

# Сортируем вершины симплекса, первая - наибольшая f
simplex0=simplex = simplex[argsort(-f(simplex))]
# По всем точкам, кроме xh ищем центр тяжести
xc=1/n*sum(array([s for s in simplex[1:]]),axis=0)

simplexs=[]
N=0
while 1/(n+1)*sum(array([abs(f(s)-f(xc)) for s in simplex]))>eps: #
Проверка сходимости
      N+=1
      if N>1e3:
            break
      # Выбираем три опорные точки
      xh,xg,xl=simplex[0],simplex[1],simplex[-1]

      # По всем точкам, кроме xh ищем центр тяжести
      xc=1/n*sum(array([s for s in simplex[1:]]),axis=0)

      # Генерируем отраженную точку
      xr=(1+alpha)*xc-alpha*xh

      Shrink=True
      if f(xr)<f(xl):
            xe=(1-gamma)*xc+gamma*xr
            if f(xe)<f(xl):
                  xh=xe
                  Shrink=False
            elif f(xe)>f(xl):
                  xh=xr
                  Shrink=False
      elif (f(xl)<f(xr))&(f(xr)<f(xg)):
            xh=xr
            Shrink=False
      elif (f(xh)>f(xr))&(f(xr)>f(xg)):
            xr,xh=xh,xr
            Shrink=True
      elif (f(xr)>f(xh)):
            Shrink=True

      if Shrink:
```

```
            xs=beta*xh+(1-beta)*xc
            if f(xs)<f(xh):
                  xh=xs
            elif f(xs)>f(xh):
                  # Обновляем симплекс и сжимаем его
                  simplex[0],simplex[1],simplex[-1]=xh,xg,xl
                  # Сжимаем симплекс, кроме xl
                  simplex[:-1]=xl+(simplex[:-1]-xl)/2
                  xh,xg,xl=simplex[0],simplex[1],simplex[-1]

      # Обновляем симплекс
      simplex[0],simplex[1],simplex[-1]=xh,xg,xl
      # Сортируем симплекс
      simplex = simplex[argsort(-f(simplex))]
      simplexs.append(simplex)

print('Extrema found by ',N,' steps, final point is
',1/(n+1)*sum(array([s for s in simplex]),axis=0))

if n==2:
      fig = plt.figure()
      ax = fig.add_subplot(111)

      ax.set_xlim(-2,-0.5)
      ax.set_ylim(-1,1)

      X, Y = np.meshgrid(np.arange(-2,2,0.1), np.arange(-2,2,0.1))
      ax.contour(X,Y,-f(np.array([Y,X]).T),100)

      poly=patches.Polygon(simplex0,closed=True, fc='b', ec='r')
      patch=ax.add_patch(poly)

      def init():
          return patch,

      def animate(i):
          poly.set_xy(simplexs[i])
          patch=ax.add_patch(poly)
          return patch,

      ani = animation.FuncAnimation(fig, animate, np.arange(1,
len(simplexs)), init_func=init,
                                    interval=200, blit=True)
      plt.show()
```

Генетичний алгоритм

Генетичний алгоритм - це, перш за все, метод багатовимірної оптимізації, т.е. метод пошуку мінімуму багатовимірної функції. Потенційно цей метод можна використовувати для глобальної оптимізації, але з цим виникають складності.

Сама суть метод полягає в тому, що ми моделюємо еволюційний процес: у нас є якась популяція (набор векторів), яка розмножується, на яку діють мутації і виробляється природний відбір на основі мінімізації цільової функції. Розглянемо детальніше ці процеси.



Що ж, раніше всього наша популяція повинна розмножуватися. Основний принцип розмноження — потомок схожий на своїх батьків. Т.е. ми повинні задати який-небудь механізм спадкування. І краще буде, якщо він буде включати елемент випадковості. Але швидкість розвитку таких систем дуже низька — різноманітність генетичного падає, народжується. Тобто значення функції перестає мінімізуватися.

Для вирішення цієї проблеми був введений механізм мутації, який полягає у випадковій заміні якихось осіб. Цей механізм дозволяє привнести що-то нове в генетичну різноманітність.
Наступний важливий механизм — селекція. Як було сказано, селекція — відбір особин (можна з тих, що тільки народилися, а можна з усіх — практика показує, що це не грає вирішальну роль), які краще мінімізують функцію.
Зазвичай вибирають тільки особу, скільки було до розмноження, щоб із епохи в епоху у нас було постійне кількість осіб у населенні. Також прийнято відбирати “щасливчиків” — якесь число осіб, які, можливо, погано мінімізують функції, але зато внесуть різноманітність у наступні покоління.

Цих трьох механізмів частіше всього недостатньо, щоб мінімізувати функцію. Так популяція вироджується — рано чи пізно локальний мінімум забиває своїм значенням всю популяцію. Коли таке відбувається, проводять процес, який називається струсом (в природі аналогії — глобальні катаклізми), коли знищується майже вся популяція, і додаються нові (випадкові) особи.

```
import sys
import numpy as np

class PSO:

    def __init__(self, particles, velocities, fitness_function,
                 w=0.8, c_1=1, c_2=1, max_iter=100, auto_coef=True):
        self.particles = particles
        self.velocities = velocities
        self.fitness_function = fitness_function

        self.N = len(self.particles)
        self.w = w
        self.c_1 = c_1
        self.c_2 = c_2
        self.auto_coef = auto_coef
        self.max_iter = max_iter


        self.p_bests = self.particles
        self.p_bests_values = self.fitness_function(self.particles)
        self.g_best = self.p_bests[0]
        self.g_best_value = self.p_bests_values[0]
        self.update_bests()

        self.iter = 0
        self.is_running = True
        self.update_coef()

    def __str__(self):
        return f'[{self.iter}/{self.max_iter}] $w$:{self.w:.3f} -
$c_1$:{self.c_1:.3f} - $c_2$:{self.c_2:.3f}'

    def next(self):
        if self.iter > 0:
            self.move_particles()
            self.update_bests()
            self.update_coef()
```

```
        self.iter += 1
        self.is_running = self.is_running and self.iter < self.max_iter
        return self.is_running

    def update_coef(self):
        if self.auto_coef:
            t = self.iter
            n = self.max_iter
            self.w = (0.4/n**2) * (t - n) ** 2 + 0.4
            self.c_1 = -3 * t / n + 3.5
            self.c_2 =  3 * t / n + 0.5

    def move_particles(self):

        # add inertia
        new_velocities = self.w * self.velocities
        # add cognitive component
        r_1 = np.random.random(self.N)
        r_1 = np.tile(r_1[:, None], (1, 2))
        new_velocities += self.c_1 * r_1 * (self.p_bests -
self.particles)
        # add social component
        r_2 = np.random.random(self.N)
        r_2 = np.tile(r_2[:, None], (1, 2))
        g_best = np.tile(self.g_best[None], (self.N, 1))
        new_velocities += self.c_2 * r_2 * (g_best  - self.particles)

        self.is_running = np.sum(self.velocities - new_velocities) != 0

        # update positions and velocities
        self.velocities = new_velocities
        self.particles = self.particles + new_velocities


    def update_bests(self):
        fits = self.fitness_function(self.particles)

        for i in range(len(self.particles)):
            # update best personal value (cognitive)
            if fits[i] < self.p_bests_values[i]:
                self.p_bests_values[i] = fits[i]
                self.p_bests[i] = self.particles[i]
```

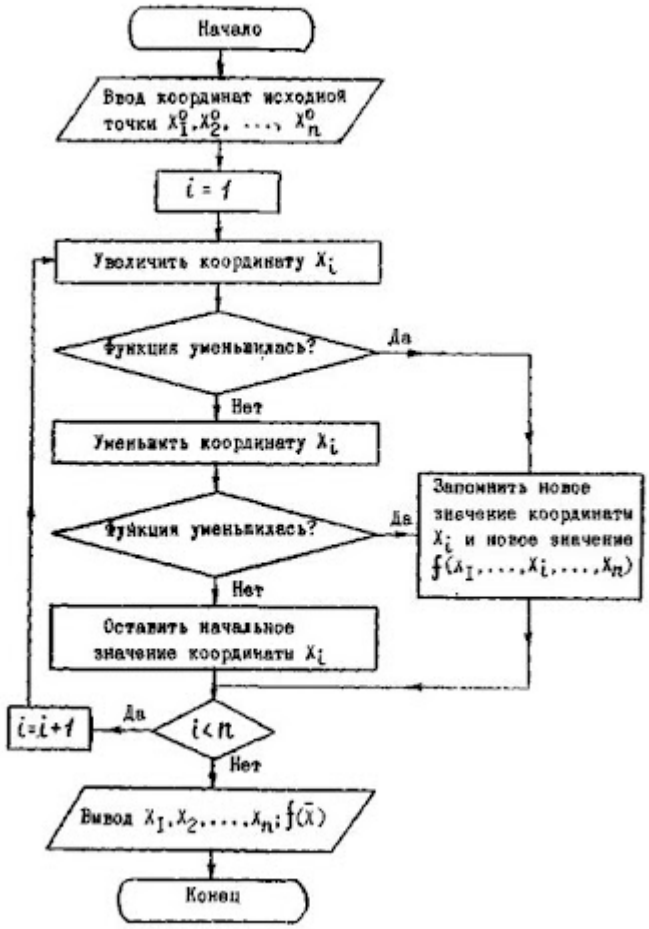
```
            # update best global value (social)
            if fits[i] < self.g_best_value:
                self.g_best_value = fits[i]
                self.g_best = self.particles[i]
```

Сенс градієнтних методів

(Градієнт - векторна величина, яка визначає в кожній точці простору не лише швидкість зміни, а й напрямок найшвидшої зміни функції, що залежить від координат

У градієнтних методах мінімізації за напрямок руху на k -й ітерації обирають вектор, протилежний градієнту функції f0 у точці xk .

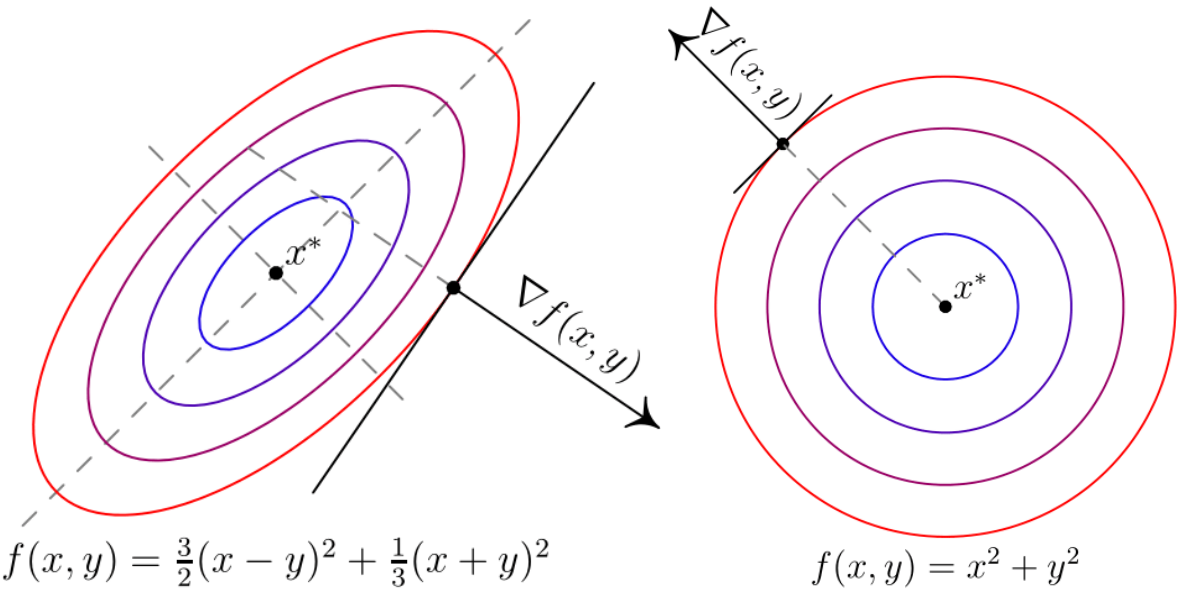
Різні варіанти градієнтного методу відрізняються способом вибору крокового множника на k -й ітерації, а також тими чи іншими способами (різницевої) апроксимації градієнтів.



Градієнтний спуск

Задача:
Перш ніж описувати метод, слід спочатку описати завдання, а саме: «Дані множина $\mathcal{K}$ і функція $f : \mathcal{K} \to \mathbb{R}$, потрібно знайти точку $x^* \in \mathcal{K}$, таку для всіх $x \in \mathcal{K}$», що зазвичай записується наприклад ось так $f(x) \to \min\limits_{x \in \mathcal{K}}$.

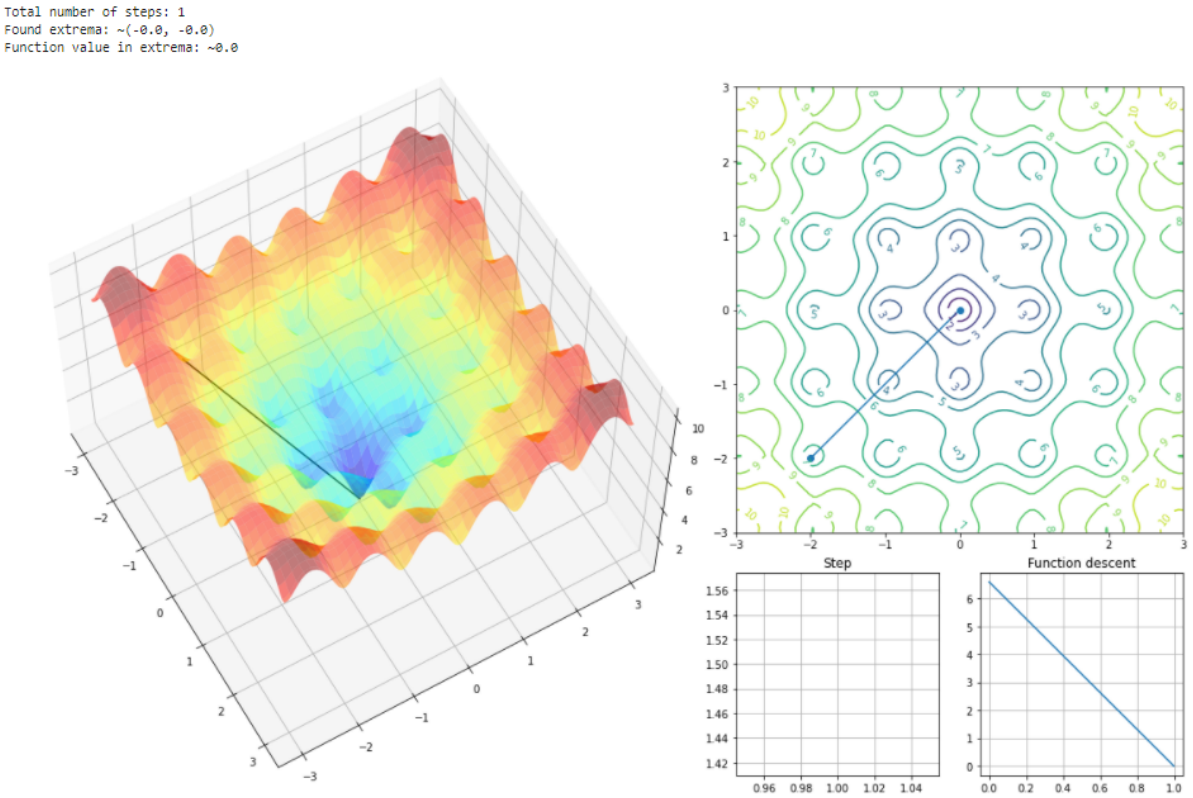
Величина $\nabla f(x^*)$— градієнт функції f у точці x^*. Також рівність градієнта нуля означає рівність всіх часткових похідних нулю, тому в багатовимірному випадку можна отримати цей критерій просто послідовно застосувавши одновимірний критерій щодо кожної змінної окремо.
Ось приклад градієнта:



Градієнтний спуск грунтується на нерівності із градієнтом:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k).$$

Ось такими собі”маленькими” кроками будемо вибирати напрямок нашого руху.



Метод спряжених градієнтів

Розглянуті вище градієнтні способи шукають точку мінімуму функції у випадку лише за нескінченне число ітерацій. Метод сполучених градієнтів формує напрямки пошуку,

більшою мірою відповідні геометрії мінімізованої функції. Це суттєво збільшує швидкість їх збіжності та дозволяє, наприклад, мінімізувати квадратичну функцію

f(x) = (x, Hx) + (b, x) + a

Алгоритм методу пов'язаних градієнтів Флетчера-Рівса полягає в наступному.

1. У точці x[0] обчислюється p[0] = -f'(x[0]).

2. На k-му кроці за наведеними вище формулами визначаються крок ak. та точка x[k+1].

3. Обчислюються величини f(x[k+1]) та f'(x[k+1]).

4. Якщо f'(x[k+1]) = 0, точка x[k+1] є точкою мінімуму функції f(x). Інакше визначається новий напрямок p[k+l] із співвідношення

$$p[k+1] = -f'(x[k+1]) + \frac{(f'(x[k+1]), f'(x[k+1]))}{(f'(x[k]), f'(x[k]))} p[k]$$

Та здійснюється перехід до наступної ітерації. Ця процедура знайде мінімум квадратичної функції не більше ніж за п кроків. При мінімізації неквадратичних функцій метод Флетчера-Рівса із кінцевого стає ітеративним. У разі після (п+1)-й ітерації процедури 1-4 циклічно повторюються із заміною x[0] на x[п+1] , а обчислення закінчуються при , де - задане число. При цьому застосовують таку модифікацію методу:

$x[k+1] = x[k] + a_k p[k]$,

$p[k] = -f'(x[k]) + b_{k-1} p[k-1]$, $k >= 1$;

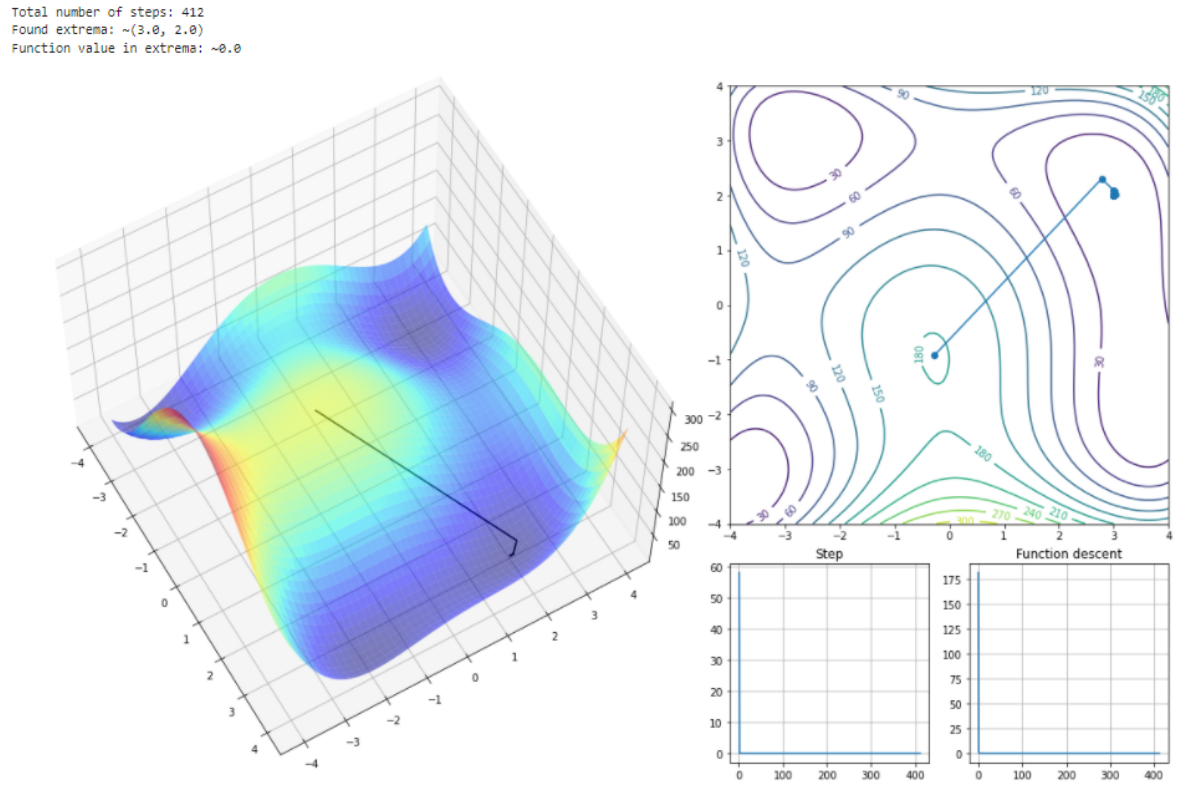
$p[0] = -f'(x[0])$;

$f(x[k] + a_k p[k]) = \min_{a \geq 0} f(x[k] + ap[k])$;

$$A^2_{k-1} = \begin{cases} \dfrac{(f'(x[k]), f'(x[k]) - f'(x[k-1]))}{(f'(x[k]), f'(x[k]))}, k \notin I \\ 0, k \in I \end{cases}$$

Тут I-множина індексів: I = {0, n, 2n, 3n, ...}, тобто оновлення методу відбувається через кожні n кроків.

Геометричний зміст методу сполучених градієнтів полягає в наступному (рис. 2.11). Із заданої початкової точки x[0] здійснюється спуск у напрямку p[0] = -f'(x[0]). У точці x[1] визначається вектор-градієнт f'(x[1]). Оскільки x[1] є точкою мінімуму функції у напрямку p[0], то f'(x[1]) ортогональний вектору p[0]. Потім знаходиться вектор p [1], H-пов'язаний до p [0] . Далі знаходиться мінімум функції вздовж напрямку p [1] і т. д.




```
"CG": self.reset,
            "GD":
    def antigradient(self, **kwargs):
        x = kwargs["x_next"]

        return - self.gradient(x)

    def reset(self, **kwargs):
        x = kwargs["x"]
        x_next = kwargs["x_next"]
        direction = kwargs["direction"]
        iteration = kwargs["iteration"]
        period = kwargs["period"]

        next_dir = None
```

```
        if 1 - ((iteration + 1) % period):
            aux1 = - self.gradient(x)
            aux2 = - self.gradient(x_next)
            coeff = np.dot(aux2.T, aux1) / np.dot(aux1.T, aux1) - 1
            next_dir = aux2 + coeff * direction
        else:
            next_dir = - self.gradient(x_next)
```

Метод градієнтів із подрібненням кроку

Задача:
Знайти argmin xεR n f0(x) для заданої неперервно диференційованої функції
f0 : Rn → R1

Загальна схема алгоритму:
Початок:
У цьому методі ми маємо вибрати довільну початкову точку x1 ∈ Rn , довільну константу ρ0 > 0 , довільний множник β ∈ [1/ 2; 1) та покласти k = 1.

Загальна схема алгоритму:
Початок:

1. У цьому методі ми маємо вибрати довільну початкову точку $x^1 \in R^n$, довільну константу $\rho_0 > 0$, довільний множник β ∈ [1/ 2; 1) та покласти $k = 1$.

Основний цикл:

2. Обчислити $\nabla f_0(x^k)$і $||\nabla f_0(x^k)||$. Якщо$\nabla f_0(x^k) = 0$, то покласти $x^* = x^k$ і зупинитися; інакше перейти на крок III.

k $\nabla f(x^k)$ III. Обчислити одиничний вектор h .

4. Покласти $\rho_k = \rho_{k-1}$.
5. Обчислити точку $x^{k+1} = x^k - \rho_k h^k$.
6. Якщо $f_0(x^{k+1}) < f_0(x^k)$, то покласти $k = k + 1$ і перейти на крок II; інакше покласти $\rho_k = \beta\rho_k$ і перейти на крок V.

Алгоритм:

Метод починає свою роботу з деякої довільно обраної точки $x1 \in Rn$, довільного множника $\beta \in [1/2;1)$, довільної константи $\rho 0>0$, та початкового $k=1$.

(1) Проводяться обчислення $\nabla f0(x^k)$ та $||\nabla f0(x^k)||$.

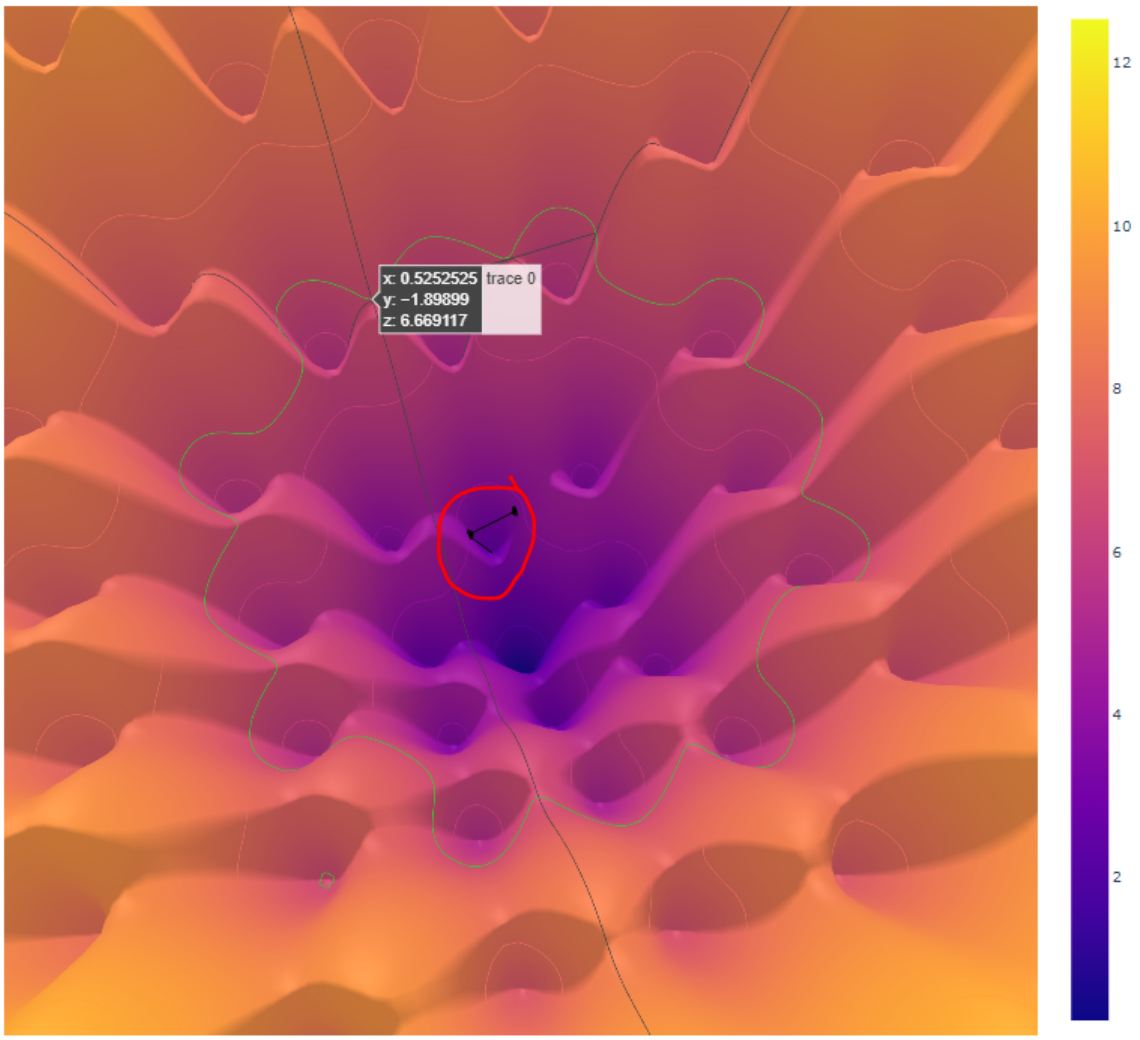
Далі за формулою $hk=\nabla f0(xk)||\nabla f0(xk)||$ обчислюється одиничний вектор.

(2) Для обчислення $xk+1$ визначимо $\rho k=\rho k-1$ та знайдемо $xk+1=xk-\rho khk$.

Перевіряємо чи справджується нерівність $f0(xk+1)<f0(xk)$ та, в разі виконання, більшуємо $k=k+1$ та переходимо на крок (1). В протилежному випадку перевизначаємо $\rho k=\beta\rho k$ та повертаємося до обчислення $xk+1$ (крок (2)).

Алгорим завершується коли $\nabla f0(xk)=0$.

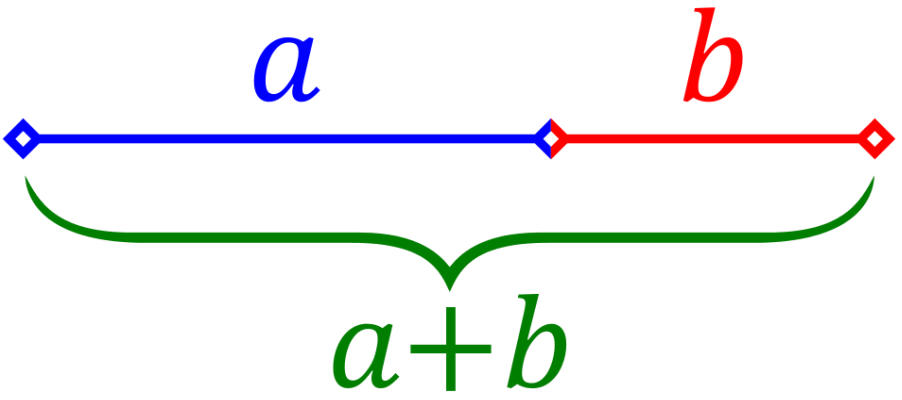
Тоді $x*=xk$.




```
def gradient_descent(self, function, start_point, p_o, beta, iteration):
        self.function = function
        self.x0 = start_point
        self.min_legend = []
        # VALIDATE USER`S INPUTS
        if p_o <= 0:
            raise Exception("p_0 should be greater than 0")
        if not ((beta < 1) and (0.5 <= beta)):
            raise Exception("beta should be in range(0.5,1)")
            # add validate on start_point(optional)

        # ALGORITHM
        # 1
        p_k = p_o
        x_k = start_point
        self.min_legend.append(x_k)
        for k in range(iteration):
```

```
            # 2
            grad_f = self.gradient(function, x_k)
            norm = np.linalg.norm(grad_f)
            if grad_f.any() == 0:
                return x_k
                break
            h_k = grad_f/norm  # 3

            p_k1 = p_k  # 4

            flag = True
            while flag:
                x_k1 = x_k-p_k*h_k  # 5
                self.min_legend.append(x_k)
                if self.function(x_k1) < self.function(x_k):
                    k = k+1
                    flag = False
                else:
                    p_k = beta*p_k  # goto 5

            x_k = x_k1
            self.min_legend.append(x_k)
        self.extr = x_k
        return x_k, self.function(x_k)
```

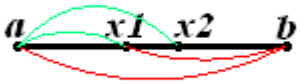
Метод золотого перерізу



$$\varphi = (a+b) : a = a : b$$

Метод золотого перерізу - метод пошуку екстремуму дійсної функції однієї змінної на заданому відрізку. В основі методу лежить принцип поділу відрізка в пропорціях золотого перетину.

Як вибрати точку?



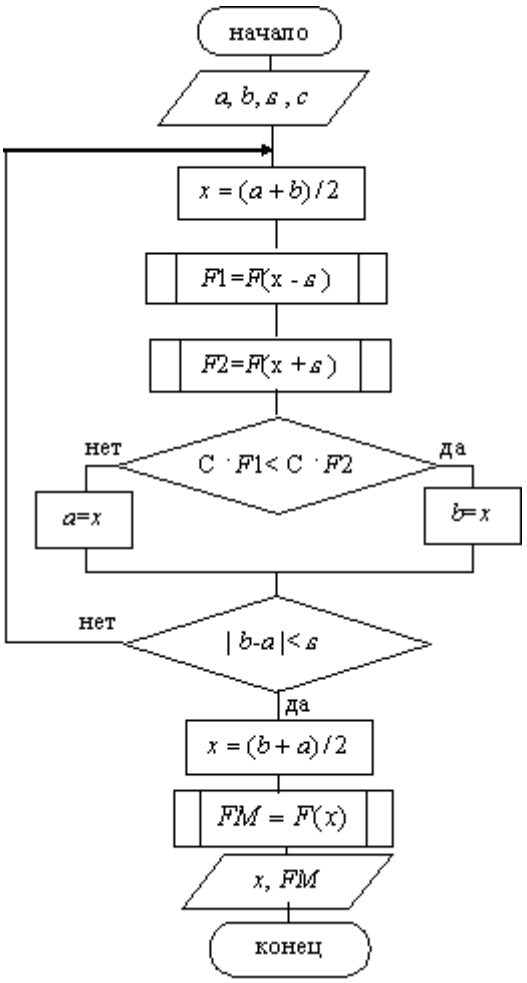
Алгоритм

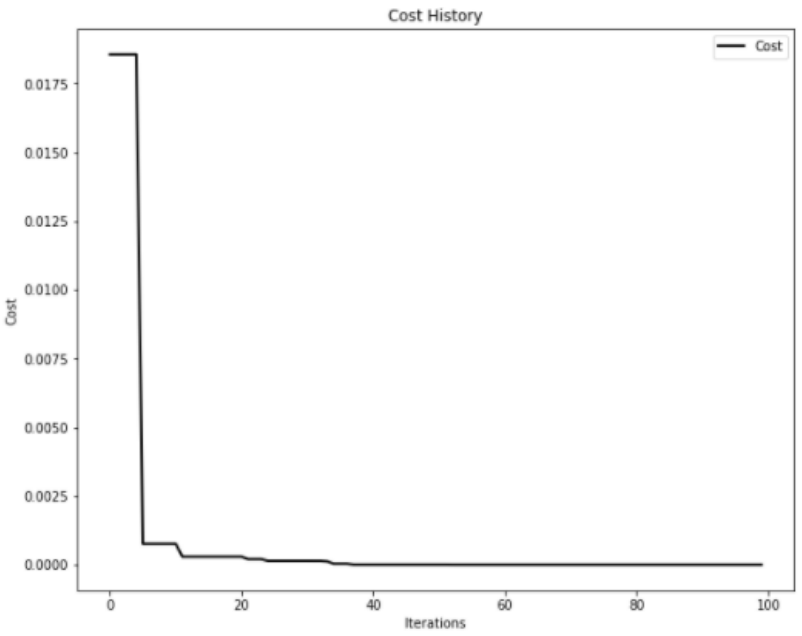
1. На першій ітерації заданий відрізок ділиться двома симетричними відносно центру точками і розраховуються значення в цих точках.
2. Після чого той з кінців відрізка, до якого серед двох знову поставлених точок ближче виявилася та, значення в якій максимальне (для випадку пошуку мінімуму), відкидають.
3. На наступній ітерації в силу показаній вище властивості золотого перетину вже треба шукати лише одну нову точку.
4. Процедура триває до тих пір, поки не буде досягнута задана точність.

Формалізація

1. **Крок 1.** Задаються початкові межі відрізка $a,\ b$ і точність ε
2. **Крок 2.** Розрахувати початкові точки поділу:

   $x_1 = b - \frac{(b-a)}{\phi}, \quad x_2 = a + \frac{(b-a)}{\phi}$ і значення в них цільової функції: $y_1 = f(x_1),\ y_2 = f(x_2)$
   - Якщо $y_1 \geq y_2$ (min search), то $a = x_1$
   - Інакше $b = x_2$
3. **Крок 3.**
   - Якщо
   - $|b-a| < \varepsilon$, то $x = \frac{a+b}{2}$ і зупинитися.
   - Інакше повернутися до кроку 2.





Падіння вартості для золотого перерізу

```
def zolot_sech(f,a,b):
    eps=0.05
    if abs(b-a)<eps:
        return (a+b)/2
    else:
        t=(b-a)/phi
        x1=b-t
        x2=a+t
        if f(x1)>=f(x2):
            return zolot_sech(f,x1,b)
        else:
            return zolot_sech(f,a,x2)
```

ClonAlg

Задача:
Розробити програмну реалізацію імунного алгоритму клонального відбору для вирішення задачі оптимізації мультимодальних функцій декількох змінних обраною мовою програмування(Python).
Задача оптимізації формулюється наступним чином: задана множина X (допустима множина задачі) і функція f(x) (цільова функція), задана на X. Необхідно знайти точки максимуму цільової функції на X. $f(x) \rightarrow \max, x \in X$

Теорія клональної селекції використовується для того, щоб пояснити, **як імунна система "бореться" проти чужорідних антигенів**. Коли бактерія проникає в наш організм, вона починає розмножуватися та вражати своїми токсинами клітини нашого організму. Була запропонована одна з теорій: Ті клітини, які здатні розпізнавати чужорідний антиген, розмножуються асексуальним способом, пропорційно до ступеня їх розпізнавання. Протягом процесу репродукції клітини окремі клітини піддаються мутації, що дозволяє їм мати більш високу відповідність до антигену, що розпізнається: чим вище афінність батьківської клітини, тим меншою мірою вони піддаються мутації, і навпаки.
 Навчання в імунній системі забезпечується збільшенням відносного розміру популяції та афінності тих лімфоцитів, які довели свою цінність під час розпізнавання представленого антигену. Основними імунними механізмами при розробці алгоритму є обробка певної множини антитіл з набору клітин пам'яті,

видалення антитіл з низькою афінністю *(сила взаємодії речовин)* , дозрівання афінності та повторний відбір клонів пропорційно їх афінності до антигенів.

Алгоритм:

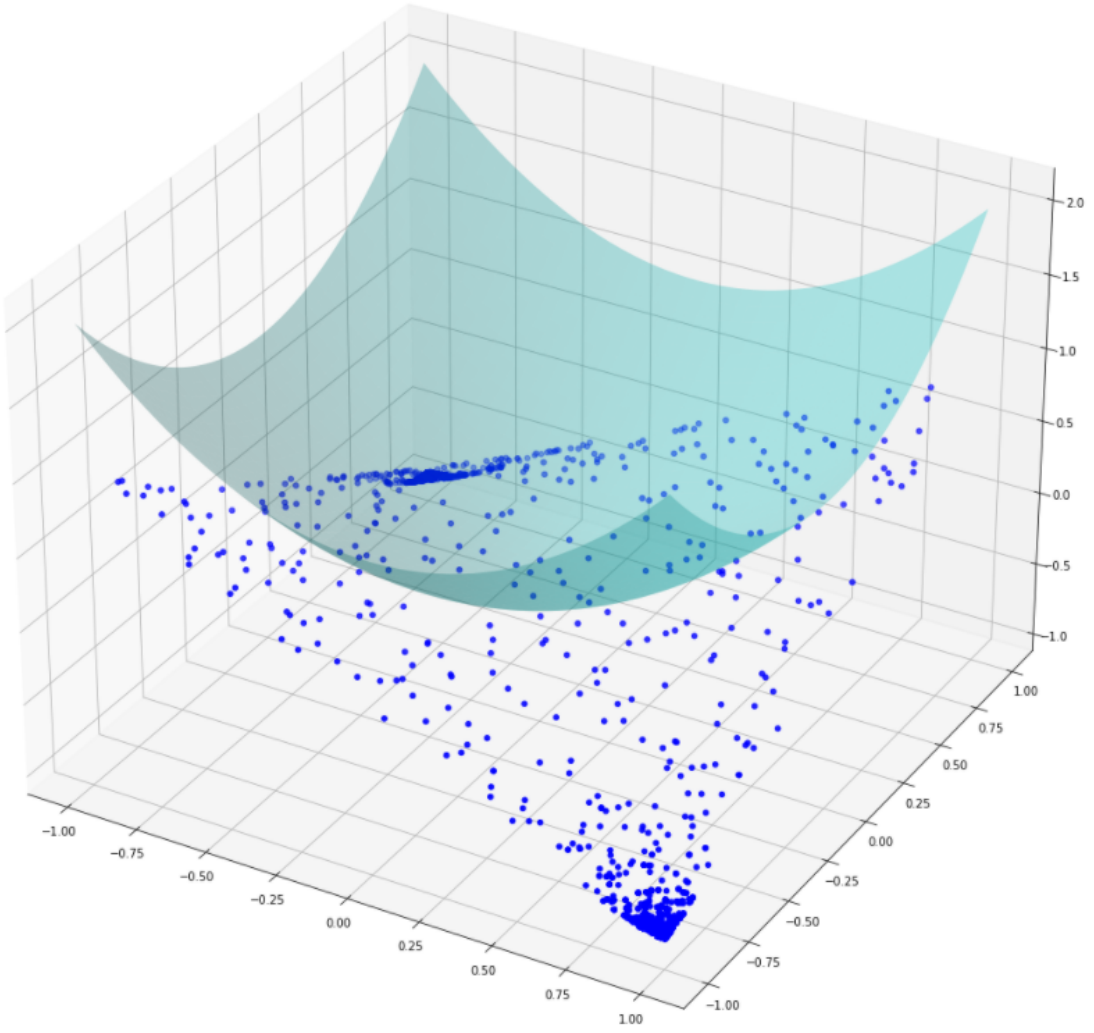
1. **Ініціалізація**: генерація випадкового початкового репертуару (популяції) атрибутів рядків (імунних клітин).
2. **Популяційний цикл**: для кожного антигену ВИКОНАТИ:

2.1. **Відбір**: відібрати клітини, що мають найвищу афінність до антигену.

2.2. **Репродукція та генетична мінливість**: створити копії імунних клітин, причому чим краще кожна клітина розпізнає антиген, тим більше створюється її копії.

2.3. **Здійснити процедуру мутації в кожній клітині інверсивно пропорційно їх афінності:** чим вища афінність, тим менший рівень мутації.

3. **Цикл:** повторювати Крок 2, поки не буде досягнуто заданий критерій зупинки




```
import numpy as np
import copy
import pylab
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import matplotlib.colors as colors
# defining test functions

def func(x):
    return (x[0] ** 2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2

def func1(x):
    return (-20.0 * math.exp(-0.2 * math.sqrt(0.5 * (x[0]**2 +
x[1]**2))) - math.exp(0.5 * (math.cos(2 * math.pi * x[0]) + math.cos(2 *
math.pi * x[1]))) + math.e + 20)
```

```
def create_antibody(quan, leng, dim):
    population = [np.random.choice([0, 1], size=(dim, leng)) for x in
range (quan)]
    return population

def convert(antibody, area):# координати - код Гріна -> к-ти - дійсні числа
    x = []
    for i in range (0,len(antibody)):
        summ=0
        for k in range (len(antibody[i])):
            summ+=antibody[i][k]*2**k
        x.append(summ)

x[i]=area[i][0]+x[i]*((area[i][1]-area[i][0])/(2**len(antibody[i])-1))
    return x

def affinity(dec_antibody,function):
    return function(dec_antibody)

def clone(population, quan_c):
    return [[copy.copy(population[i]) for x in range (quan_c)] for i in
range(len(population))]

def mutate(clones, prob):
    num = int(prob*len(clones[0][0][0]))
    for s in range(len(clones)):
        for i in range(len(clones[s])):
            for n in range (0,len(clones[s][i])):
                el = np.random.choice(range(0,len(clones[s][i][n])),num,
replace=False)
                for p in el:
                    clones[s][i][n][p]=int(not(clones[s][i][n][p]))
    return clones

def select(clones,aff):
    final_clones=[clones[i][aff[i].index(min(aff[i]))] for i in
range(len(clones))]
    return final_clones

def replace(population, final_clones, aff_ant, aff_c):
    for i in range(len(population)):
        if aff_ant[i]>aff_c[i]:
            population[i]=final_clones[i]
            aff_ant[i]=aff_c[i]

def edit(population,d,func,leng,dim,aff_ant):
```

```
    for i in range(d):
        num = aff_ant.index(max(aff_ant))
        del(aff_ant[num])
        del(population[num])
    population+=create_antibody(d, leng, dim)
def clon_alg(func, quan, leng, dim, area, prob, quan_c, gen, d):
    population=create_antibody(quan, leng, dim)
    for number in range(0,gen):
        con_ant=[]
        for i in range(quan):
            con_ant.append(convert(population[i],area))
            for k in range(dim):
                xdot[k].append(con_ant[i][k])
        aff_ant=[affinity(con_ant[i],func) for i in range(quan)]
        zdot.extend(aff_ant)
        clones = mutate(clone(population,quan_c),prob)
        aff_c=[[affinity(convert(clones[s][i],area),func) for i in
range(quan_c)] for s in range(quan)]
        clones=select(clones,aff_c)
        for i in range(quan):
            aff_c[i]=min(aff_c[i])
        replace(population, clones, aff_ant, aff_c)
        edit(population,d,func,leng,dim,aff_ant)
    ans=population[aff_ant.index(min(aff_ant))]
    return convert(ans, area)
xdot=[]
[xdot.append([]) for i in range(2)]
zdot=[]
clon_alg(func4, 100, 22, 2, [[-1,1],[-1,1]], 0.3, 10, 50, 5)
```

Висновки

У цій роботі нами були розглянуті алгоритми оптимізації за допомогою градієнтних методів, алгоритму ClonAlg, генетичного алгоритму, випадкового пошуку, алгоритму Нелдера Міда . Для цього використовувалися 2 функції, призначені для перевірки алгоритмів:

• Функція Химмельблау

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2.$$

• Функція Еклі

$$f(x, y) = -20\exp\left[-0.2\sqrt{0.5\left(x^2 + y^2\right)}\right] - \exp[0.5\left(\cos 2\pi x + \cos 2\pi y\right)] + e + 20$$

В процесі виконання лабораторної роботи нами були написані алгоритми оптимізації функцій двох змінних із перевіркою критеріїв зупинки та підрахунком кількості кроків, візуалізація роботи алгоритмів та функцій мовою Python.