МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проєктування

Звіт
про виконання лабораторної роботи №1
з дисципліни «методи оптимізції»

“Дослідження методів одновимірного пошуку на основі виключення інтервалів”

Виконав:
студент 2 курсу, групи КН-32
Сафонов Дмитро
Володимирович

Київ – 2025

Варіант 20

Мета роботи: Ознайомитись і дослідити методи пошуку оптимальних параметрів та методи знаходження інтервалу невизначеності для функцій з однією змінною. Набути навичок реалізації методів виключення інтервалів, порівняти час роботи декількох методів, реалізувати метод Девіса-Свена-Кемпі (ДСК).

Завдання:

20	$G(x) = \sin^4 x + 6(x-1)^2 + 10$	0	2

Хід роботи:

1. **Візуалізувати сформовану функцію в межах інтервалу невизначеності:**

Побудуємо графік ф-ї за допомогою wolfram mathematica:

```
Plot[Sin[x]^4 + 6 * (x - 1)^2 + 10, {x, 0, 2}]
граф  синус
```

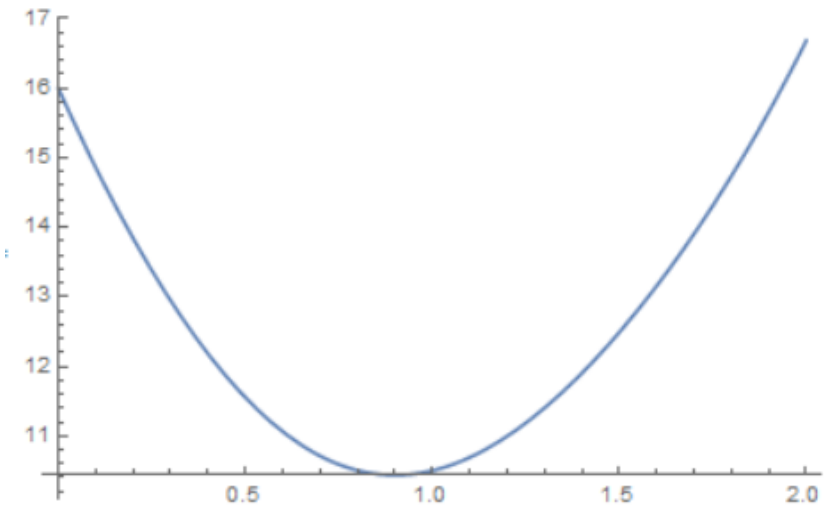

Рис 1. – графік функції

На графіку видно, що на проміжку $[0; 2]$ функція є унімодальною.

2. **Використати похідну для визначення мінімуму функції**

2.1. Використати похідну першого порядку для визначення точки екстремуму шляхом розв'язання рівняння $G'(x) = 0$.

Для цього також скористаємося wolfram mathematica:

```
In[2]:= D[Sin[x]^4 + 6 * (x - 1)^2 + 10, x]
        |·· |синус

Out[2]= 12 (-1 + x) + 4 Cos[x] Sin[x]^3

In[4]:= NSolve[12 (x - 1) + 4 Cos[x] Sin[x]^3 == 0, x, Reals]
        |чисельне наближення роз··· |косинус |синус |множина

Out[4]= {{x → 0.900367}}
```

Отже мінімумом функціє є точка $x = 0.900367$.

2.2. Використати похідну другого порядку для підтвердження, що знайдена точка являється саме мінімумом функції.

```
In[13]:= D[Sin[x]^4 + 6 * (x - 1)^2 + 10, {x, 2}]
         |·· |синус

Out[13]= 12 + 12 Cos[x]^2 Sin[x]^2 - 4 Sin[x]^4

In[14]:= f2[x_] := 12 + 12 Cos[x]^2 Sin[x]^2 - 4 Sin[x]^4
                   |косинус |синус |синус

         f2[0.9004]

Out[15]= 13.3361
```

Друга похідна є додатною у точці x, отже x – мінімум функції.

3. Для наступних методів з пунктів 4-6:

- Заміряти час роботи методу (за необхідності використати велику кількість викликів).
- Порахувати Кількість Обчислень Цільової Функції (КОЦФ).
- Для 4 та 6 використати Точність пошуку: $\varepsilon = 0.001$.

4. Реалізувати метод простого перебору (брутфорс) для пошуку точки мінімуму функції на початковому інтервалі невизначеності.

Лістинг методу грубої сили:

```cpp
#include <iostream>
#include <cmath>
#include <chrono>

using namespace std;
using namespace chrono;

int function_calls = 0;

double G(double x)
{
    function_calls++;
    return pow(sin(x), 4) + 6 * pow(x - 1, 2) + 10;
}

int main()
{
    double a, b, eps, min, xmin;
    a = 0;
    b = 2;
    eps = 0.001;
    min = 100;
    xmin = 0;

    auto start = high_resolution_clock::now();

    for (double i = a; i <= b; i += eps)
    {
        double value = G(i);

        if (value < min)
        {
            min = value;
            xmin = i;
        }
    }

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);

    cout << "x min = " << xmin << endl;
    cout << "G(x min) = " << min << endl;
    cout << "Execution time: " << duration.count() << " us" << endl;
    cout << "Number of function calls = " << function_calls << endl;

    return 0;
}
```

Даний алгоритм знаходить всі значення функції з вказаним кроком на визначеному інтервалі

Для вимірювання часу використовується бібліотека <chrono>, яка дозволяє зручно вимірювати час у потрібних одиницях.

Для підрахунку КОЦФ використовується змінна function_calls.

Результати методу грубої сили для заданої точності 0.001:

x min = 0.9

G(x min) = 10.4365

Execution time: 338 us

Number of function calls = 2001

Результат методу грубої сили для заданої точності 0.0001:

x min = 0.9004

G(x min) = 10.4365

Execution time: 3367 us

Number of function calls = 20001

5. **Реалізувати метод знаходження потенційно кращого інтервалу невизначеності Девіса-Свена-Кемпі.**
   - За стартову точку можна обрати одну з точок початкового інтервалу невизначеності.
   - Провести пошук з 4 початковими кроками (можна додати свої): $h1 = 10{-}3, h2 = 10{-}2, h3 = 10{-}1, h4 = 1$.
   - Для кожного пошуку порахувати КОЦФ.

Лістинг для методу Девіса-Свена-Кемпі (DSK):

```
#include <iostream>
#include <cmath>
#include <chrono>

using namespace std;
using namespace chrono;

int function_calls = 0;

double G(double x)
{
    function_calls++;
    return pow(sin(x), 4) + 6 * pow(x - 1, 2) + 10;
}
```

```
void DSK(double x0, double h, double* a, double* b, int max_iterations)
{
    double x1, x2, x3, g0, g1, g2, g3;
    x1 = x0 - h;
    x2 = x0 + h;
    g0 = G(x0);
    g1 = G(x1);
    g2 = G(x2);

    int iterations = 0;

    bool move_left = (g1 < g0);

    while (iterations < max_iterations)
    {
        if (g0 < g1 && g0 < g2) // мінімум між x1 та x2 - ідеальний випадок
        {
            h /= 4;
            if (move_left) // рухалися вліво
            {
                x3 = x1 + h;
                g3 = G(x3);

                if (g3 < g0)
                {
                    x2 = x0;
                }
                else if (g0 < g3)
                {
                    x1 = x3;
                }

            }
            else // рухалися вправо
            {
                x3 = x2 - h;
                g3 = G(x3);

                if (g0 < g3)
                {
                    x2 = x3;
                }
                else if (g3 < g0)
                {
                    x1 = x0;
                }
            }

            *a = x1;
            *b = x2;
            return;
        }
        else if (g1 < g0 && g2 < g0) // умова унімодальності порушена
        {
            cout << "The function is not unimodal" << endl;
            return;
        }
        else if (g1 < g0) // рухаємося в бік x1
        {
            x2 = x0;
            x0 = x1;
            g2 = g0;
            g0 = g1;
```

```
            x1 = x0 - h;
            g1 = G(x1);
        }
        else if (g2 < g0) // рухаємося в бік x2
        {
            x1 = x0;
            x0 = x2;
            g1 = g0;
            g0 = g2;
            x2 = x0 + h;
            g2 = G(x2);
        }

        h *= 2;
        iterations++;

        if (iterations >= max_iterations)
        {
            cout << "Max iterations reached in DSK method" << endl;
            return;
        }
    }
}


int main()
{
    double a, b, eps, h, x0, x1, x2, g0, g1, g2;
    a = 0;
    b = 2;

    x0 = a;
    h = 1;

    auto start = high_resolution_clock::now();

    DSK(x0, h, &a, &b, 100);

    auto stop = high_resolution_clock::now();
    auto duration_micro = duration_cast<microseconds>(stop - start);
    auto duration_nano = duration_cast<nanoseconds>(stop - start);

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "Execution time: " << duration_micro.count() << " us" << endl;
    cout << "Execution time: " << duration_nano.count() << " ns" << endl;
    cout << "Number of function calls = " << function_calls << endl;

    return 0;
}
```

Результати виконання алгоритму ДСК для різних кроків h:

Для $h = 1$:

a = 0

b = 1.5

Execution time: 1 us

Execution time: 1900 ns

Number of function calls = 5

Інтервал був покращений. Для цього знадобилося 5 разів викликати функцію. Алгоритм виконувався майже 2 мкс.

Для h = 0.1:

a = 0.4

b = 1.2

Execution time: 2 us

Execution time: 2600 ns

Number of function calls = 8

Інтервал став ще вужчий. Часу та викликів функції знадобилося більше.

Для h = 0.01:

a = 0.64

b = 1.28

Execution time: 2 us

Execution time: 2800 ns

Number of function calls = 11

Інтервал став ще вужчий. Часу та викликів функції знадобилося більше.

Для h = 0.001:

a = 0.512

b = 1.536

Execution time: 4 us

Execution time: 3200 ns

Number of function calls = 15

Для h = 0.0001:

a = 0.4096

b = 1.2288

Execution time: 3 us

Execution time: 3700 ns

Number of function calls = 18

Провівши тестування роботи алгоритму для різних кроків, було отримано такі результати:

Крок h	Час (нс)	a	b	КОЦФ
1	1900	0	1.5	5
0.1	2600	0.4	1.2	8
0.01	2800	0.64	1.28	11
0.001	3200	0.512	1.536	15
0.0001	3700	0.4096	1.2288	18

Проаналізувавши результати, можно зробити висновок, що зменшення кроку не гарантує звуження остаточного інтервалу, але точно збільшує КОФЦ та кількість затраченого часу. Найбільш оптимальним для даної функції показав себе кроки $h = 0.01$ та $h = 0.1$, оскільки вони визачили найвужчий інтервал за мінімальну кількість часу та кількість обчислень цільової функції.

У алгоритмі також реалізовано оптимізацію, яка пізля визначення інтервалу, на якому точно знаходиться мінімум, звужує його на 25%. У оптимізації лежить ідея тернарного пошуку.

6. **Реалізувати метод дихотомії для пошуку точки мінімуму функції, а також за бажанням інші методи виключення інтервалів:**

   - Метод тернарного пошуку
   - Метод золотого перетину
   - Метод Фібоначчі

   Виконати пошук для початкового інтервалу невизначеності та для всіх знайдених інтервалів з пункту 4, порахувати КОЦФ, обрати найбільш оптимальний початковий крок $hopt$ для методу ДСК.

   Метод дихотомії

   Лістинг:

```
#include <iostream>
#include <cmath>
#include <chrono>

using namespace std;
using namespace chrono;

int function_calls = 0;

double G(double x)
{
    function_calls++;
    return pow(sin(x), 4) + 6 * pow(x - 1, 2) + 10;
}

double dichotomy(double a, double b, double eps)
{
    double l, xm, x1, x2, gm, g1, g2, xmin;
    l = b - a;
    xm = (a + b) / 2;
    gm = G(xm);

    while (b - a > eps)
    {
        x1 = a + l / 4;
        x2 = b - l / 4;
        g1 = G(x1);
        g2 = G(x2);
        if (g1 < gm)
        {
            b = xm;
            xm = x1;
            gm = g1;
        }
        else if (g2 < gm)
        {
            a = xm;
            xm = x2;
            gm = g2;
        }
        else
```

```
        {
            a = x1;
            b = x2;
        }
        l = b - a;
    }

    return xmin = (a + b) / 2;
}


int main()
{
    double a, b, eps, xmin;
    a = 0;
    b = 2;
    eps = 0.001;

    auto start = high_resolution_clock::now();

    xmin = dichotomy(a, b, eps);

    auto stop = high_resolution_clock::now();
    auto duration_micro = duration_cast<microseconds>(stop - start);
    auto duration_nano = duration_cast<nanoseconds>(stop - start);

    cout << "x min = " << xmin << endl;
    cout << "G(x min) = " << G(xmin) << endl;
    cout << "Execution time: " << duration_micro.count() << " us" << endl;
    cout << "Execution time: " << duration_nano.count() << " ns" << endl;
    cout << "Number of function calls = " << function_calls << endl;

    return 0;
}
```

Результати для різних проміжків:

Початковий проміжок $[0; 2]$:

x min = 0.900391

G(x min) = 10.4365

Execution time: 4 us

Execution time: 4600 ns

Number of function calls = 24

Проміжки знайдені за допомогою DSK-алгоритму:

Для $h = 1$, проміжок $[0; 1.5]$:

x min = 0.900513

G(x min) = 10.4365

Execution time: 4 us

Execution time: 4300 ns

Number of function calls = 24

Для h = 0.1 проміжок [0.4; 1.2]:

x min = 0.900391

G(x min) = 10.4365

Execution time: 4 us

Execution time: 4300 ns

Number of function calls = 22

Для h = 0.01 проміжок [0.64; 1.28]:

x min = 0.900313

G(x min) = 10.4365

Execution time: 4 us

Execution time: 4100 ns

Number of function calls = 22

Для h = 0.001 проміжок [0.512; 1.536]:

x min = 0.9005

G(x min) = 10.4365

Execution time: 4 us

Execution time: 4100 ns

Number of function calls = 22

Для h = 0.0001 проміжок [0.4096; 1.2288]:

x min = 0.9004

G(x min) = 10.4365

Execution time: 4 us

Execution time: 4200 ns

Number of function calls = 22

Результати обчислень для різних початкових проміжків:

Проміжок	Значення мінімуму	Час (нс)	КОЦФ
[0; 2]	0.900391	4600	24
[0; 1.5]:	0.900513	4300	24
[0.4; 1.2]	0.900391	4300	22
[0.64; 1.28]	0.900313	4100	22
[0.512; 1.536]	0.9005	4100	22
[0.4096; 1.2288]	0.9004	4200	22

Аналізуючи результати, можна сказати, що оптимальніший початковий інтервал не дає значного виграшу для методу дихотомії. Більш точний початковий інтервал може знизити КОЦФ, але незначно і для методу дихотомії попереднє використання DSK-алгоритму не є раціональним. Оскільки, для прикладу, взявши крок 0.1 в DSK-алгоритмі ми обчислюємо цільову функцію 8 разів, а в методі дихотомії кількість обчислень зменшується лише на 2. Тобто ми виконали 6 зайвих обчислень. Покрщення у часі також непомітні.

Метод золотого перетину:

Лістинг:

```
#include <iostream>
#include <cmath>
#include <chrono>

using namespace std;
using namespace chrono;

int function_calls = 0;

double G(double x)
{
    function_calls++;
    return pow(sin(x), 4) + 6 * pow(x - 1, 2) + 10;
}
```

```
double golden_ratio(double a, double b, double eps)
{
    double l, x1, x2, g1, g2, xmin;

    l = b - a;
    x1 = a + 0.382 * l;
    x2 = a + 0.618 * l;
    g1 = G(x1);
    g2 = G(x2);


    while (b - a > eps)
    {
        if(g1 > g2)
        {
            a = x1;
            l = b - a;
            x1 = x2;
            g1 = g2;
            x2 = a + 0.618 * l;
            g2 = G(x2);
        }
        else if (g1 < g2)
        {
            b = x2;
            l = b - a;
            x2 = x1;
            g2 = g1;
            x1 = a + 0.382 * l;
            g1 = G(x1);
        }
        else
        {
            a = x1;
            b = x2;
            l = b - a;
            x1 = a + 0.382 * l;
            x2 = a + 0.618 * l;
            g1 = G(x1);
            g2 = G(x2);
        }
    }

    return xmin = (a + b) / 2;
}


int main()
{
    double a, b, eps, xmin;
    a = 0;
    b = 2;
    eps = 0.001;

    auto start = high_resolution_clock::now();

    xmin = golden_ratio(a, b, eps);

    auto stop = high_resolution_clock::now();
    auto duration_micro = duration_cast<microseconds>(stop - start);
    auto duration_nano = duration_cast<nanoseconds>(stop - start);
```

```
    cout << "x min = " << xmin << endl;
    cout << "G(x min) = " << G(xmin) << endl;
    cout << "Execution time: " << duration_micro.count() << " us" << endl;
    cout << "Execution time: " << duration_nano.count() << " ns" << endl;
    cout << "Number of function calls = " << function_calls << endl;

    return 0;
}
```

Результати для різних проміжків:

Початковий проміжок $[0; 2]$:

x min = 0.900321

G(x min) = 10.4365

Execution time: 4 us

Execution time: 4300 ns

Number of function calls = 19

Проміжки знайдені за допомогою DSK-алгоритму:

Для $h = 1$, проміжок $[0; 1.5]$:

x min = 0.90052

G(x min) = 10.4365

Execution time: 4 us

Execution time: 4400 ns

Number of function calls = 19

Для $h = 0.1$ проміжок [0.4; 1.2]:

x min = 0.900452

G(x min) = 10.4365

Execution time: 3 us

Execution time: 3500 ns

Number of function calls = 17

Для h = 0.01 проміжок [0.64; 1.28]:

x min = 0.900451

G(x min) = 10.4365

Execution time: 3 us

Execution time: 3600 ns

Number of function calls = 17

Для h = 0.001 проміжок [0.512; 1.536]:

x min = 0.900328

G(x min) = 10.4365

Execution time: 3 us

Execution time: 3900 ns

Number of function calls = 18

Для h = 0.0001 проміжок [0.4096; 1.2288]:

x min = 0.900505

G(x min) = 10.4365

Execution time: 3 us

Execution time: 3500 ns

Number of function calls = 17

Результати обчислень для різних початкових проміжків:

Проміжок	Значення мінімуму	Час (нс)	КОЦФ
[0; 2]	0.900321	4300	19
[0; 1.5]:	0.90052	4400	19
[0.4; 1.2]	0.900452	3500	17
[0.64; 1.28]	0.900451	3600	17
[0.512; 1.536]	0.900328	3900	18
[0.4096; 1.2288]	0.900505	3500	17

Використання DSK-алгоритму показало такий самий результат, що і для методу дихотомії. Можна виконати на 2 менше КОЦФ, але при цьому виконати значно більше обчислень при покращенні інтервалу.

7. **Занести у зведену таблицю всі результати по часу, загальному значенню КОЦФ та знайденим точкам мінімуму.**

7.1. Порівняти метод простого перебору та методи виключення інтервалів. 7.2.

7.2. Порівняти метод виключення інтервалів з використанням початкового інтервалу невизначеності, та з використанням методу ДСК та уточнених інтервалів невизначеності.

Порівняння методів:

Метод	Час (нс)	КОЦФ	Ефективність
Брутфорс	3300 мкс	20001	Найгірший
Дихотомія	4600	24	Хороший
Золотий перетин	4300	19	Найкращий
Дихотомія + DSK	3700-4300	22-24	Хороший
Золотий перетин + DSK	3500-4400	17-19	Хороший

Затрати на DSK-алгоритм нівелюють переваги, які він дає методам дихотомії та золотого перетину (на проміжку [0;2] та заданої точності 0.001). Для більшої точності, наприклад, 0.000001 DSK-алгоритм виграшу також не дає, що логічно, оскільки саме на перших кроках метод дихотомії та золотого перетину відкидають найбільші фрагменти інтервалу.

Порівняння КОЦФ для різної точності:

Метод	eps = 0.1	eps = 0.01	eps = 0.001
Брутфорс	20	200	2001
Дихотомії	12	18	24
Золотого перетину	10	15	19

Метод золотого перетину дозволяє виконувати меншу КОЦФ, у порівнянні з методом дихотомії, що робить його більш оптимальним.

Висновок: під час виконання цієї лабораторної роботи я ознайомився з такими методами виключення інтервалів: метод дихотомії, метод тернарного пошуку, метод золотого перетину, метод Фібоначі. Також дізнався про ДСК-алгоритм, який використовується для звуження початкового інтервалу, і реалізував його. Реалізував метод дихотомії та метод золотого перетину. Дізнався про особливості кожного з методів, провівши тестування з різною точністю, різними початковими інтервалами.

Після аналізу результатів методів дихотомії та золотого перетину прийшов до висновку, що ДСК-алгоритм не доцільно використовувати для даних методів. ДСК-алгоритм використовують для визначення початкового інтервалу, на якому точно буде мінімум. Можливо через те, що в умові завдання був заданий досить вдалий інтервал [0;2] ефективність ДСК-алгоритму здається невеликою. Але якщо початковий інтервал невідомий або є дуже великим, то ДСК-алгоритм буде дуже корисним.