***Звіт***

*Програмування інтелектуальних інформаційних систем*

Лабораторна робота №6

“Реалізувати метод пошуку по багатограннику для знаходження мінімуму функції”

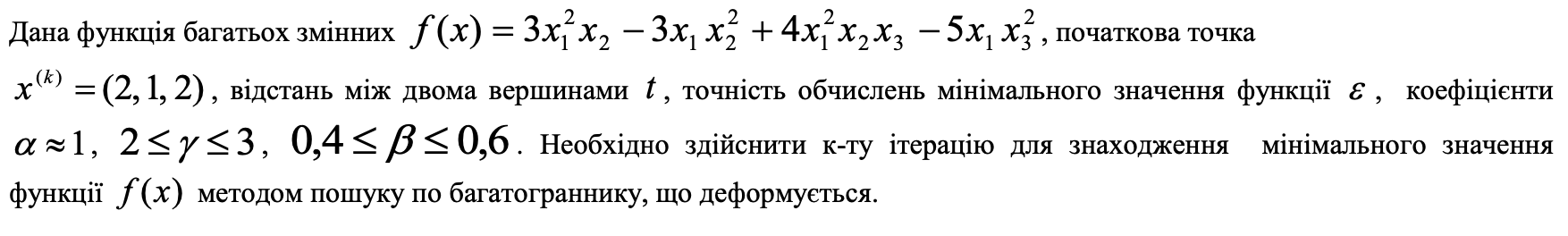
Студентки групи ІП-01

Галько Міли Вячеславівни

**Лабораторна робота №6**

“Реалізувати метод пошуку по багатограннику для знаходження мінімуму функції”

**Варіант №6**

****

**Реалізація завдання**

Маємо реалізувати клас для взаємодії і утримання матриці – SimplexMatrix.

**Код**

using System.Collections;  
using System.Text;  
  
namespace Lab6;  
  
public struct SimplexMatrix  
**{** public double this[int row, int column]  
 {  
 get => \_matrix[row, column];  
 set => \_matrix[row, column] = value;  
 }  
  
 public int Height => \_matrix.GetLength(0);  
 public int Width => \_matrix.GetLength(1);  
  
 private readonly double[,] \_matrix;  
   
 public SimplexMatrix(int rows, int columns)  
 {  
 \_matrix = new double[rows, columns];  
 }  
  
 public double[] GetRow(int m)  
 {  
 var arr = new double[Width];  
 for (int **i** = 0; **i** < Width; **i**++)  
 {  
 arr[**i**] = this[m, **i**];  
 }  
  
 return arr;  
 }  
   
 public void SetRow(int m, double[] row)  
 {  
 for (int **i** = 0; **i** < Width; **i**++)  
 {   
 this[m, **i**] = row[**i**];  
 }  
 }  
  
 public override string ToString()  
 {  
 var strBuilder = new StringBuilder();  
 for (int **i** = 0; **i** < Height; **i**++)  
 {  
 strBuilder.Append(' ');  
 strBuilder.AppendJoin(", ", GetRow(**i**));  
 strBuilder.Append('\n');  
 }  
  
 return strBuilder.ToString();  
 }  
**}**

Далі для відтворення методу потребуємо передати у нього вхідні значення. Нехай клас методу пошуку по багатограннику для знаходження мінімуму – NelderMead. І в нього будемо передавати параметри: цільова функція (TargetFunction), початкова точка (initial), відстань між вершинами (1), точність обчислення (0,01) та кількість ітерацій (300).

**Код**

static double TargetFunction(double[] vector)  
**{** return  
 + 3 \* vector[0] \* vector[0] \* vector[1]  
 - 3 \* vector[0] \* vector[1] \* vector[1]  
 + 4 \* vector[0] \* vector[0] \* vector[1] \* vector[2]  
 - 5 \* vector[0] \* vector[2] \* vector[2];  
**}**

var initial = new double[] { 2, 1, 2 };  
var nelderMead = new NelderMead(initial, 1);   
double[] functionValues = new double[nelderMead.SimplexTable.Height];  
  
UpdateFunctionValues();  
  
nelderMead.Apply(TargetFunction, 300, 0.01);   
  
UpdateFunctionValues();  
  
  
void UpdateFunctionValues()  
{  
 for (int **row** = 0; **row** < functionValues.Length; **row**++)  
 {  
 functionValues[**row**] = TargetFunction(nelderMead.SimplexTable.GetRow(**row**));  
 }  
  
 Array.Sort(functionValues);  
}

Інші параметри задаються у класі NelderMead. Також він утримує передані дані та методи реалізації пошуку по багатограннику, і додатковий метод взаємодії із SimplexMatrix.

**Код**

namespace Lab6;  
  
public class NelderMead  
**{** private const double **Alpha** = 1;  
 private const double **Beta** = 0.5;  
 private const double **Gamma** = 2.5;  
 private const double **Delta** = 0.5;  
  
 private SimplexMatrix \_simplexTable;  
 private double[] \_functionValues;  
 private int[] \_indexes;  
  
 public SimplexMatrix SimplexTable => \_simplexTable;  
  
 public NelderMead(double[] initialVector, double distanceBetweenTwoPoints)  
 {  
 GenerateSimplexMatrix(initialVector, distanceBetweenTwoPoints);  
 }  
  
 public void Apply(Func<double[], double> function, int iterationCount, double precision)  
 {  
 \_functionValues = new double[\_simplexTable.Height];  
 \_indexes = new int[\_simplexTable.Height];  
   
 for (int **i** = 0; **i** < iterationCount; **i**++)  
 {  
 for (int **row** = 0; **row** < \_functionValues.Length; **row**++)  
 {  
 \_functionValues[**row**] = function(\_simplexTable.GetRow(**row**));  
 \_indexes[**row**] = **row**;  
 }  
  
 Array.Sort(\_functionValues, \_indexes);  
 Array.Reverse(\_functionValues);  
 Array.Reverse(\_indexes);  
 //Sorting by descending  
  
 double maxFuncValue = \_functionValues[0];  
 double secMaxFuncValue = \_functionValues[1];  
 double minFuncValue = \_functionValues[^1];  
  
 if (!double.IsFinite(maxFuncValue) || !double.IsFinite(minFuncValue))  
 {  
 return;  
 }  
  
 int indexMax = \_indexes[0];  
 int indexMin = \_indexes[^1];  
 double[] maxRow = \_simplexTable.GetRow(indexMax);  
 double[] centroid = new double[\_simplexTable.Width];  
   
 for (int **row** = 0; **row** < \_simplexTable.Height; **row**++)  
 {  
 if (**row** == indexMax)  
 continue;  
  
 AddRow(centroid,\_simplexTable, **row**);  
 }  
 DivideByFactor(centroid,\_simplexTable.Width);  
  
 if (Math.Sqrt(\_functionValues  
 .Select(functionValue => Math.Pow(functionValue - function(centroid), 2))  
 .Sum() / (\_simplexTable.Height)) <= precision)  
 {  
 break;  
 }  
  
 double[] reflected = new double[\_simplexTable.Width];  
 for (int **row** = 0; **row** < reflected.Length; **row**++)  
 {  
 reflected[**row**] = centroid[**row**] + **Alpha** \* (centroid[**row**] - maxRow[**row**]);  
 }  
  
 double reflectedFuncValue = function(reflected);  
 if (reflectedFuncValue < secMaxFuncValue &&  
 reflectedFuncValue >= minFuncValue)  
 {  
 \_simplexTable.SetRow(indexMax, reflected);  
 continue;  
 }  
  
 if (reflectedFuncValue < minFuncValue)  
 {  
 double[] expandedPoint = new double[\_simplexTable.Width];  
 for (int **row** = 0; **row** < expandedPoint.Length; **row**++)  
 {  
 expandedPoint[**row**] = centroid[**row**] + **Gamma** \* (reflected[**row**] - centroid[**row**]);  
 }  
  
 \_simplexTable.SetRow(indexMax,  
 function(expandedPoint) <= reflectedFuncValue ? expandedPoint : reflected);  
 continue;  
 }  
  
 double[] contracted = new double[\_simplexTable.Width];  
 if (reflectedFuncValue >= secMaxFuncValue)  
 {  
 for (int **row** = 0; **row** < contracted.Length; **row**++)  
 {  
 contracted[**row**] = centroid[**row**] + **Beta** \* (reflected[**row**] - centroid[**row**]);  
 }  
  
 if (function(contracted) <= maxFuncValue)  
 {  
 \_simplexTable.SetRow(indexMax, contracted);  
 continue;  
 }  
 }  
  
 var minRow = \_simplexTable.GetRow(indexMin);  
 for (var **j** = 0; **j** < \_simplexTable.Height; **j**++)  
 {  
 if (**j** == indexMin) continue;  
  
 var currentRow = \_simplexTable.GetRow(**j**);  
 for (int **k** = 0; **k** < currentRow.Length; **k**++)  
 {  
 currentRow[**k**] = minRow[**k**] + **Delta** \* (currentRow[**k**] - minRow[**k**]);  
 }  
  
 \_simplexTable.SetRow(**j**, currentRow);  
 }  
 }  
 }  
  
 private static void DivideByFactor(double[] arr, double divider)  
 {  
 for (int **i** = 0; **i** < arr.Length; **i**++)  
 {  
 arr[**i**] /= divider;  
 }  
 }  
  
 private static void AddRow(double[] arr, in SimplexMatrix mat, int row, double factor = 1)  
 {  
 for (int **i** = 0; **i** < arr.Length; **i**++)  
 {  
 arr[**i**] += mat[row, **i**] \* factor;  
 }  
 }  
  
 private void GenerateSimplexMatrix(double[] initial, double twoPointsDistance)  
 {  
 \_simplexTable = new SimplexMatrix(initial.Length + 1, initial.Length);  
 int N = \_simplexTable.Width;  
 for (int **i** = 0; **i** < N; **i**++)  
 {  
 \_simplexTable[0, **i**] = initial[**i**];  
 }  
  
 for (int **i** = 1; **i** < \_simplexTable.Height; **i**++)  
 {  
 for (int **j** = 0; **j** < N; **j**++)  
 {  
 if (**j** == **i** - 1)  
 {  
 \_simplexTable[**i**, **j**] = \_simplexTable[0, **j**] + D1();  
 }  
 else  
 {  
 \_simplexTable[**i**, **j**] = \_simplexTable[0, **j**] + D2();  
 }  
 }  
 }  
  
 double D1() => twoPointsDistance / (N \* Math.Sqrt(2)) \* (Math.Sqrt(N + 1) + N - 1);  
  
 double D2() => twoPointsDistance / (N \* Math.Sqrt(2)) \* (Math.Sqrt(N + 1) - 1);  
 }  
**}**

**Результати**

