МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

«Харківський Політехнічний Інститут»

Кафедра стратегічного управління

ЗВІТ

з лабораторної роботи №4

з дисципліни «Дослідження операцій»

Перевірив

Лисицький В. Л.

Виконав:

ст. гр. КН-27

Харків – 2019

**Ціль:** Вивчити методи безумовної оптимізації другого порядку. Скласти блок-схему алгоритму вибраного метода: метод Левенберга-Марквардта. Скласти програму, що реалізує алгоритм вибраного метода. Мінімізувати функцію, що була запропонована.

**Завдання:** Варіант №2.

Формула

Початковий вектор

(0, 1)

Алгоритм методу:

**Крок 1**.Вибрати , максимальну кількість ітерацій М, покласти k=0 та .

**Крок 2**. Обчислити .

**Крок 3**. Якщо або k M, то останов. Інакше перейти до кроку 4.

**Крок 4**. Обчислити -1*f (E –* одинична матриця*)*.

**Крок 5**. Покласти .

**Крок 6.** Якщо *f() < f()*, та йти до кроку 7. Інакше до кроку 8.

**Крок 7**. Покласти та йти до кроку 2.

**Крок 8**. Покласти , та йти до кроку 4.

**Хід роботи**

1. Блок-схему алгоритму наведено на рисунку 1.

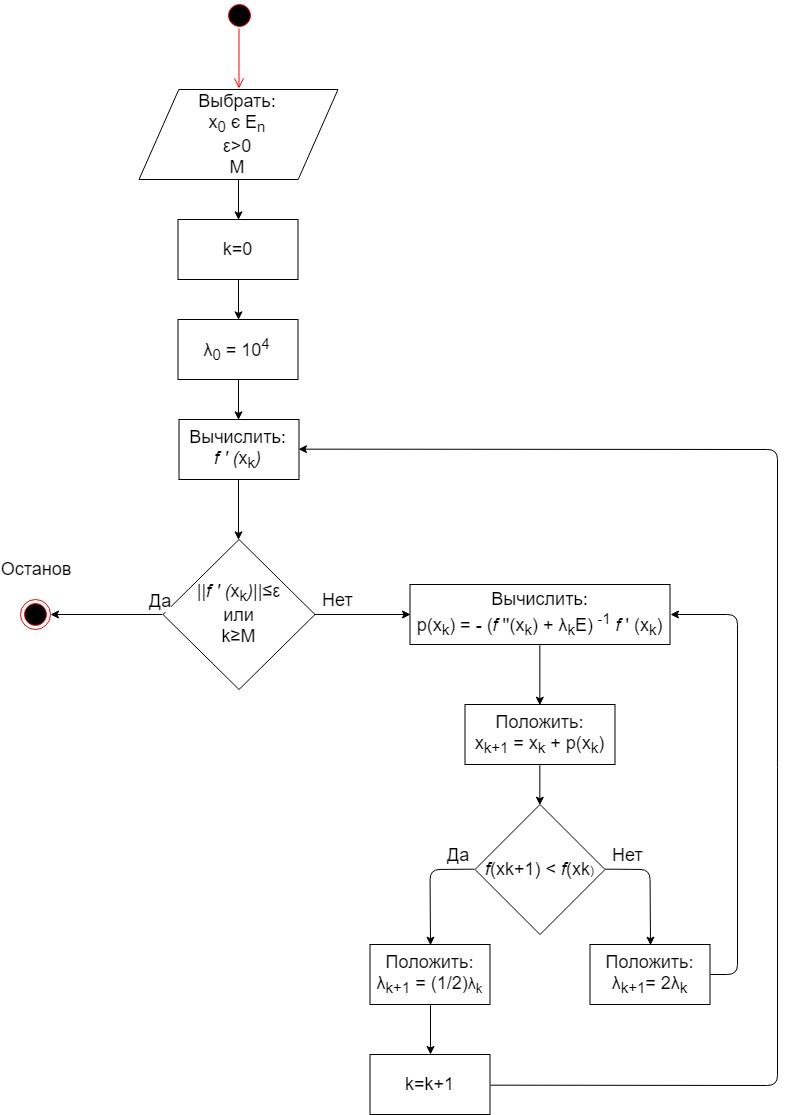


Рисунок 1 – Блок схема алгоритму Левенберга-Марквардта

1. Реалізацію у JS наведено нижче.

Файл lab4.js

|  |
| --- |
| "use strict"  let log = ""  document.addEventListener("DOMContentLoaded", () => {      let eps = document.getElementById('epsilon').value      let x\_start = document.getElementById('start').value.split(',').map(Number)        let input1 = document.getElementById('epsilon')      input1.addEventListener("change", () =>{          eps = input1.value;      })      let input2 = document.getElementById('start')      input2.addEventListener("change", () =>{          x\_start = input2.value.split(',').map(Number)      })      document.getElementById('run').addEventListener('click', () => Levenberg(eps, x\_start))  });  const eps = 0.000001  // const myFunction = x\_vector => 2 \* Math.pow(x\_vector[0], 2) + 2 \* Math.pow(x\_vector[1], 2) + 2 \* x\_vector[0] \* x\_vector[1] - 4 \* x\_vector[0] - 6 \* x\_vector[1]  const myFunction = x\_vector => (x\_vector[0] - x\_vector[1])\*\*2 + ((x\_vector[0]+x\_vector[1]-10)/3)\*\*2  function findDerivative(func,x\_vector,num){      let arr\_second = [...x\_vector]      arr\_second[num] += eps      return (func(arr\_second)-func(x\_vector))/eps  }  function gradient (x\_vector){      let grad = []      for (let i = 0; i < x\_vector.length; i++){          grad.push(findDerivative(myFunction, x\_vector, i))      }      return grad  }  const getScalar = (vector1, vector2) => vector1.reduce((total, value, index, array) => total + value\*vector2[index],0)  function matrixVectorMul(matrix, p\_k){      let res = new Array(p\_k.length).fill(0)      for (let row = 0; row < matrix.length; row++){          for (let col = 0; col < matrix[row].length; col++){              res[col] += matrix[row][col] \* p\_k[row]          }      }      return res  }  function getStep(x\_k,p\_k){      return -getScalar(gradient(x\_k),p\_k)/getScalar(p\_k, matrixVectorMul(secondDerivativeMatrix(myFunction,x\_k), p\_k))  }  function getNorm(x\_vector){      return Math.pow(x\_vector.reduce((t,v,i,a) => t+v\*v,0),1/2)  }  function getSum(a,b){      return a.map((v,i,a) => v += b[i])  }  function secondDerivativeMatrix(func,x\_k){      let matrix = []      let grad = gradient(x\_k)      for (let i = 0; i < x\_k.length; i++){          let x\_k\_new = [...x\_k]          x\_k\_new[i] += eps          let gradI = gradient(x\_k\_new)          let row = []          for (let j = 0; j < x\_k.length; j++){              row.push((gradI[j] - grad[j])/eps)          }          matrix.push(row)      }      return matrix  }  function createUnitMatrix(size){      let um = []      for (let i = 0; i < size; i++){          let string = []          for (let j = 0; j < size; j++){              string.push(j==i ? 1 : 0)          }          um.push(string)      }      return um  }  function scalarMatrixMul(scalar, matrix){      let result = []      for (var i = 0; i < matrix.length; i++)          result[i] = matrix[i].slice();      for (let i = 0; i < result[0].length; i++){          for (let j = 0; j < result[0].length; j++){              result[i][j] \*= scalar          }      }      return result  }  function getInverseMatrix(matrix){      let inversed = []      return inversed  }  function matricesSum(m1, m2){      for (let i = 0; i < m1[0].length; i++){          for (let j = 0; j < m1[0].length; j++){              m2[i][j] += m1[i][j]          }      }      return m2  }  function Levenberg(e1, x\_start){      let p = []      let x = [x\_start]      let um = createUnitMatrix(x\_start.length)      // Step 1      let max\_iterations = 100      let eps = 0.0001      let k = 0      // let l = [10\*\*4]      let l\_k = 10\*\*4      let l\_k\_1 = 10\*\*4      let x\_k = [...x\_start]      let x\_k\_1 = [...x\_start]      let p\_k = []      while(true){          // Step 2          let grad = gradient([...x\_k])          console.log("grad " + grad)          // Step 3          if (getNorm(grad)<eps || k > max\_iterations){              break          }          while(true){              console.log("step : " + l\_k)              console.log("k : " + k)              log += "<br>Итерация " + (k+1) + "<br>"              // Step 4              p\_k = matrixVectorMul(scalarMatrixMul(-1,(matrix\_invert(matricesSum(secondDerivativeMatrix(myFunction, x\_k), scalarMatrixMul(l\_k,um))))), gradient(x\_k))              // Step 5              x\_k\_1 = x\_k.map((el, i) => el + p\_k[i])              log += ("lambda: " + l\_k) + "<br>"              log += ("p: " + p\_k.map(el => el.toFixed(4))) + "<br>"              log += ("x: " + x\_k.map(el => el.toFixed(4))) + "<br>"              log += ("x\_k+1: " + x\_k\_1.map(el => el.toFixed(4))) + "<br>"              log += ("z(x\_k+1): " + myFunction(x\_k\_1).toFixed(4)) + "<br>"              log += ("z(x): " + myFunction(x\_k).toFixed(4)) + "<br>"              // Step 6              k++              if (myFunction(x\_k\_1) < myFunction(x\_k)){                  l\_k = 1/2 \* l\_k                  x\_k = x\_k\_1                  break              }              else{                  l\_k \*= 2                  x\_k = x\_k\_1              }          }      }      log+='Iterations: '+k      console.log('Iterations: '+k)      document.getElementById("answer").innerHTML = log  } |

1. Приклад розрахунків програми.

|  |
| --- |
| Итерация 1  lambda: 10000  p: 0.0004,0.0000  x: 0.0000,1.0000  x\_k+1: 0.0004,1.0000  z(x\_k+1): 9.9984  z(x): 10.0000  Итерация 2  lambda: 5000  p: 0.0008,0.0000  x: 0.0004,1.0000  x\_k+1: 0.0012,1.0000  z(x\_k+1): 9.9952  z(x): 9.9984  Итерация 3  lambda: 2500  p: 0.0016,0.0000  x: 0.0012,1.0000  x\_k+1: 0.0028,1.0000  z(x\_k+1): 9.9888  z(x): 9.9952  Итерация 4  lambda: 1250  p: 0.0032,0.0000  x: 0.0028,1.0000  x\_k+1: 0.0060,1.0000  z(x\_k+1): 9.9761  z(x): 9.9888  Итерация 5  lambda: 625  p: 0.0064,0.0000  x: 0.0060,1.0000  x\_k+1: 0.0123,1.0000  z(x\_k+1): 9.9508  z(x): 9.9761  Итерация 6  lambda: 312.5  p: 0.0126,0.0001  x: 0.0123,1.0000  x\_k+1: 0.0250,1.0002  z(x\_k+1): 9.9008  z(x): 9.9508  Итерация 7  lambda: 156.25  p: 0.0249,0.0006  x: 0.0250,1.0002  x\_k+1: 0.0499,1.0007  z(x\_k+1): 9.8032  z(x): 9.9008  Итерация 8  lambda: 78.125  p: 0.0485,0.0022  x: 0.0499,1.0007  x\_k+1: 0.0983,1.0029  z(x\_k+1): 9.6169  z(x): 9.8032  Итерация 9  lambda: 39.0625  p: 0.0921,0.0080  x: 0.0983,1.0029  x\_k+1: 0.1904,1.0109  z(x\_k+1): 9.2751  z(x): 9.6169  Итерация 10  lambda: 19.53125  p: 0.1676,0.0281  x: 0.1904,1.0109  x\_k+1: 0.3580,1.0391  z(x\_k+1): 8.6872  z(x): 9.2751  Итерация 11  lambda: 9.765625  p: 0.2863,0.0884  x: 0.3580,1.0391  x\_k+1: 0.6443,1.1275  z(x\_k+1): 7.7561  z(x): 8.6872  Итерация 12  lambda: 4.8828125  p: 0.4514,0.2338  x: 0.6443,1.1275  x\_k+1: 1.0957,1.3613  z(x\_k+1): 6.3924  z(x): 7.7561  Итерация 13  lambda: 2.44140625  p: 0.6635,0.4985  x: 1.0957,1.3613  x\_k+1: 1.7592,1.8598  z(x\_k+1): 4.5342  z(x): 6.3924  Итерация 14  lambda: 1.220703125  p: 0.8888,0.8116  x: 1.7592,1.8598  x\_k+1: 2.6480,2.6714  z(x\_k+1): 2.4348  z(x): 4.5342  Итерация 15  lambda: 0.6103515625  p: 0.9964,0.9760  x: 2.6480,2.6714  x\_k+1: 3.6444,3.6474  z(x\_k+1): 0.8149  z(x): 2.4348  Итерация 16  lambda: 0.30517578125  p: 0.8022,0.7994  x: 3.6444,3.6474  x\_k+1: 4.4466,4.4469  z(x\_k+1): 0.1360  z(x): 0.8149  Итерация 17  lambda: 0.152587890625  p: 0.4120,0.4118  x: 4.4466,4.4469  x\_k+1: 4.8587,4.8587  z(x\_k+1): 0.0089  z(x): 0.1360  Итерация 18  lambda: 0.0762939453125  p: 0.1207,0.1207  x: 4.8587,4.8587  x\_k+1: 4.9793,4.9793  z(x\_k+1): 0.0002  z(x): 0.0089  Итерация 19  lambda: 0.03814697265625  p: 0.0190,0.0190  x: 4.9793,4.9793  x\_k+1: 4.9984,4.9984  z(x\_k+1): 0.0000  z(x): 0.0002  Итерация 20  lambda: 0.019073486328125  p: 0.0016,0.0016  x: 4.9984,4.9984  x\_k+1: 4.9999,4.9999  z(x\_k+1): 0.0000  z(x): 0.0000  Iterations: 20 |

1. Отримали наступне рішення
2. Керівництво користувача

Точність та початковий вектор вводимо у відповідних полях – рисунок 2

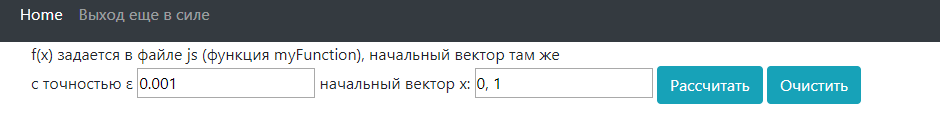


Рисунок 2 – Налаштування параметрів

Функцію вводимо у файлі lab4.js на зразок вже введеної – рисунок 3



Рисунок 3 – Налаштування функції для пошуку

Висновки

При виконанні лабораторної роботи було реалізовано алгоритм Левенберга – Марквардта. Цей алгоритм являється вдосконаленням методу Ньютона, при якому зменшується ризик розбіжності множини {xk}.