Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования Московский государственный технический университет имени Н.Э. Баумана

Лабораторная работа №5. Вариант 1.
«Методы поиска условного экстремума»
по курсу
«Методы оптимизации»

Студент группы ИУ9-82 Белогуров А.А.

Преподаватель Каганов Ю.Т.

Москва, 2018

Содержание

1 Цель работы

1. Изучение алгоритмов условной оптимизации.
2. Разработка программ реализации алгоритмов условной оптимизации.
3. Нахождение оптимальных условий решений для задач с учетом ограничений.

2 Постановка задачи

Дано: 1 Вариант. Функция Розенброка на множестве R^2:

$$f(x) = \sum_{i=1}^{n-1} [a(x_i^2 - x_{i+1})^2 + b(x_i - 1)^2] + f_0, \tag{1}$$

где

$$a = 50, \quad b = 2, \quad f_0 = 10, \quad n = 2, \tag{2}$$

тогда функция $f(x)$ будет выглядеть следующим образом:

$$f(x) = 50 * (x_0^2 - x_1)^2 + 2 * (x_0 - 1)^2 + 10 \tag{3}$$

Функции ограничений:

$$\begin{cases} g_1(x_1, x_2) = x_1^2 + x_2^2 - 1 \leq 0 \\ g_2(x_1, x_2) = -x_1 \leq 0 \\ g_3(x_1, x_2) = -x_2 \leq 0 \end{cases} \tag{4}$$

1. Найти условный экстремум методами:
   (a) Штрафных функций;
   (b) Барьерных функций;
   (c) Модифицированных функций Лагранжа;
   (d) Проекции градиента.
2. Найти все стационарные точни и значения функций, соотвествующие этим точкам.
3. Оценить скорость сходимости указанных алгоритмов.
4. Реализовать алгоритмы с помощью языка программирования высокого уровня.

3 Исследование

Найдем глобальные экстремумы функции

$$f(x) = 50(x_0^2 - x_1)^2 + 2(x_0 - 1)^2 + 10 \tag{5}$$

с помощью сервиса WolframAlpha.com:

$$min(f(x)) = 10, \quad (x_0, x_1) = (1, 1) \tag{6}$$

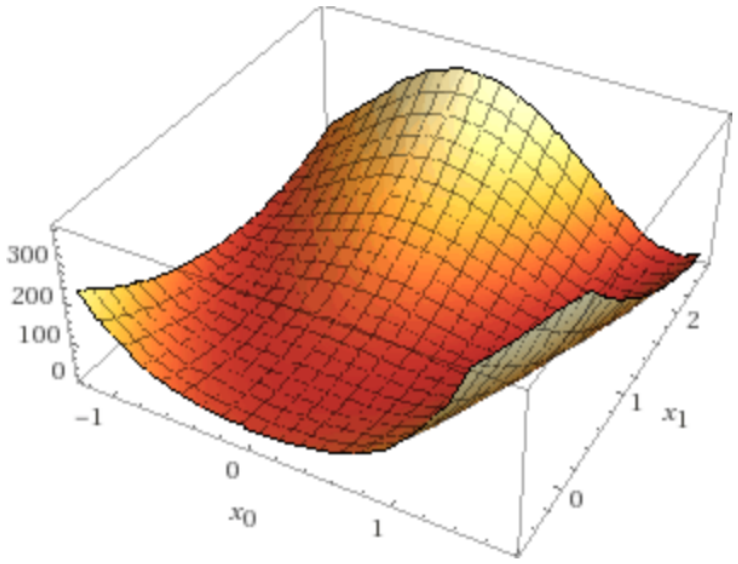

Рис. 1: График функции $f(x)$

3.1 Метод щтрафных функций.

Идея метода заключается в сведении задачи на условный минимум к решению последовательности задач поиска безусловного минимума вспомогательной функции:

$$F(x, r^k) = f(x) + P(x, r^k) \to \min_{x \in R^n}, \tag{7}$$

где $P(x, r^k)$ - штрафная функция, r^k - параметр штрафа, задаваемый на каждой k- й итерации. Это связано с возможностью применения эффективных и надежных методов поиска безусловного экстремума,

3.2 Метод барьерных функций.

Идея метода заключается в сведении задачи на условный минимум к решению последовательности задач поиска безусловного минимума вспомогательной функции:

$$F(x, r^k) = f(x) + P(x, r^k) \to \min_{x \in R^n}, \tag{8}$$

где $P(x, r^k)$ - штрафная функция, r^k - параметр штрафа. Используется обратная штрафная функция $P(x, r^k) = -r^k \sum_{j=1}^{m} \frac{1}{g_i(x)}$.

3.3 Метод модифицированных функций Лагранжа.

Стратегия аналогична используемой в методе внешних штрафов, только штрафная функция добавляется не к целевой функции, а к классической функции Лагранжа. В результате задача на условный минимум сводится к решению последовательности задач поиска безусловного минимума модифицированной функции Лагранжа:

$$L(x, \lambda^k, \mu^k, r^k) = f(x) + \sum_{j=1}^{l} \lambda_j^k g_j(x) + \frac{r^k}{2} \sum_{j=1}^{l} g_j^2(x) + \frac{1}{2r^k} \sum_{j=l+1}^{m} (max^2(0, \mu_j^k + r^k g_j(x)) - \mu^2) \tag{9}$$

где λ^k - векторы множителей Лагранжа; r^l - параметр штрафа; k - номер итерации.

3.4 Метод проекции градиента.

Стратегия поиска решения задачи учитывает тот факт, что решение x^* может лежать как внутри, так и на границе множества допустимых решений. Для определения приближенного решения x^* строится последовательность точек

$$\{x^*\} : x^{k+1} = x^k + \delta x^k, \quad k = 1, .., \tag{10}$$

где приращение δx^k определяется в каждой точке x^k в зависимости от того, где ведется поиск – внутри или на границе множества допустимых решений.

4 Практическая реализация

Все методы были реализованы на языке программирования **Kotlin**.

Листинг 1. Метод штрафных функций.

```
fun penaltyFunction(xStart: List<Double>,
                    eps: Double,
                    penaltyCoef: Double,
                    increaseParam: Double,
                    function: (xValues: Matrix<Double>) -> Double,
                    gradient: (xValues: Matrix<Double>) -> Matrix<Double>,
                    condFunctions: (xValues: Matrix<Double>) ->
                     ↪  List<Double>): Matrix<Double> {
    PrintUtils.printInfoStart("Penalty Function")

    var xPoint = create(xStart.toDoubleArray())
    var k = 0
    var penaltyValue = 0.0
    var currentPenaltyCoef = penaltyCoef

    do {
        fun penaltyFun(xValues: Matrix<Double>): Double {
            var functionValue = 0.0
            for (i in 0 until 3) {
                functionValue += currentPenaltyCoef / 2.0 *
                 ↪  maxCondFunctions(i, xValues, condFunctions).pow(2)
            }
            return functionValue
        }

        fun penaltyFun2(xValues: Matrix<Double>): Double {
            var functionValue = function(xValues)
            for (i in 0 until 3) {
                functionValue += currentPenaltyCoef / 2.0 *
                 ↪  maxCondFunctions(i, xValues, condFunctions).pow(2)
            }
            return functionValue
        }

        val directionsStep = listOf(eps, eps)
        val minimum = hookeJeeves(directionsStep, eps, ::penaltyFun2,
         ↪  gradient)

        penaltyValue = penaltyFun(minimum)

```

```
        xPoint = minimum
        currentPenaltyCoef *= increaseParam
        k += 1

    } while (penaltyValue.absoluteValue > eps)

    PrintUtils.printInfoEndFunction(k, 0, xPoint, function)
    return xPoint
}
```

Листинг 2. Метод барьерных функций.

```
fun barrierFunction(xStart: List<Double>,
                    eps: Double,
                    penaltyCoef: Double,
                    decreaseParam: Double,
                    function: (xValues: Matrix<Double>) -> Double,
                    gradient: (xValues: Matrix<Double>) -> Matrix<Double>,
                    condFunctions: (xValues: Matrix<Double>) ->
                     ↪  List<Double>): Matrix<Double> {
    PrintUtils.printInfoStart("Barrier Function")

    var xPoint = create(xStart.toDoubleArray())
    var k = 0
    var barrierValue = 0.0
    var currentPenaltyCoef = penaltyCoef

    do {
        fun barrierFun(xValues: Matrix<Double>): Double {
            var functionValue = 0.0
            for (i in 0 until 3) {
                functionValue -= currentPenaltyCoef / maxCondFunctions(i,
                 ↪  xValues, condFunctions)
            }
            return functionValue
        }

        fun barrierFun2(xValues: Matrix<Double>): Double {
            var functionValue = function(xValues)
            for (i in 0 until 3) {
                functionValue += currentPenaltyCoef / maxCondFunctions(i,
                 ↪  xValues, condFunctions)
            }
            return functionValue
        }

```

```
        val directionsStep = listOf(eps, eps)
        val minimum = hookeJeeves(directionsStep, eps, ::barrierFun2,
          ↪   gradient)

        barrierValue = barrierFun(minimum)

        xPoint = minimum
        currentPenaltyCoef /= decreaseParam
        k += 1

    } while (barrierValue.absoluteValue > eps)

    PrintUtils.printInfoEndFunction(k, 0, xPoint, function)
    return xPoint
}
```

Листинг 3. Метод модифицированных функций Лагранжа.

```
fun lagrangeFunctions(xStart: List<Double>,
                      eps: Double,
                      penaltyCoef: Double,
                      increaseParam: Double,
                      function: (xValues: Matrix<Double>) -> Double,
                      gradient: (xValues: Matrix<Double>) -> Matrix<Double>,
                      condFunctions: (xValues: Matrix<Double>) ->
                        ↪   List<Double>): Matrix<Double> {
    PrintUtils.printInfoStart("Lagrange  Functions")

    val lagrangeMu = mat[10.pow(-3), 10.pow(-3)]

    var k = 0
    var xPoint = create(xStart.toDoubleArray())
    var currentPenaltyCoef = penaltyCoef
    var lagrangeValue = 0.0

    do {
        fun lagrangeFun(xValues: Matrix<Double>): Double {
            var funValue = function(xValues)
            for (i in 0 until 3) {
                funValue += 1 / 2 / currentPenaltyCoef * ((max(0.0,
                  ↪   condFunctions(lagrangeMu)[i] + currentPenaltyCoef *
                  ↪   maxCondFunctions(i, xValues, condFunctions))).pow(2) -
                  ↪   condFunctions(lagrangeMu)[i].pow(2))
            }
            return funValue
        }
```

```
25
26          fun lagrangeFun1(xValues: Matrix<Double>): Double {
27              var funValue = 0.0
28              for (i in 0 until 3) {
29                  funValue += 1 / 2 / currentPenaltyCoef * ((max(0.0,
                        ↪   condFunctions(lagrangeMu)[i] + currentPenaltyCoef *
                        ↪   maxCondFunctions(i, xValues, condFunctions))).pow(2) -
                        ↪   condFunctions(lagrangeMu)[i].pow(2))
30              }
31              return funValue
32          }
33
34          val directionsStep = listOf(eps, eps)
35          val minimum = hookeJeeves(directionsStep, eps, ::lagrangeFun,
              ↪   gradient)
36
37          lagrangeValue = lagrangeFun1(minimum)
38
39          for (i in 0 until 2) {
40              lagrangeMu[i] = max(0.0, condFunctions(lagrangeMu)[i] +
                    ↪   currentPenaltyCoef * condFunctions(minimum)[i])
41          }
42
43          currentPenaltyCoef *= increaseParam
44          xPoint = minimum
45
46          k += 1
47
48      } while (lagrangeValue > eps)
49
50      PrintUtils.printInfoEndFunction(k, 0, xPoint, function)
51      return xPoint
52  }
```

Листинг 4. Метод проекции градиента.

```
1   fun gradientProjections(xStart: List<Double>,
2                           eps: Double,
3                           function: (xValues: Matrix<Double>) -> Double,
4                           derivFunction: (xValues: Matrix<Double>) ->
                              ↪   List<Double>,
5                           condFunctions: (xValues: Matrix<Double>) ->
                              ↪   List<Double>,
6                           derivCondFunction: (xValues: Matrix<Double>) ->
                              ↪   List<List<Double>>): Matrix<Double> {
7       PrintUtils.printInfoStart("Gradient Projections")
```

```
8
9       val k = 0
10      var xPoint = create(xStart.toDoubleArray())
11
12      var at_atinv: Matrix<Double> = create(doubleArrayOf())
13      var deltaX = create(doubleArrayOf(), numCols = 2, numRows = 1)
14      var deltaXNorm = 0.0
15
16      do {
17          if (k > maxIterations) {
18              PrintUtils.printInfoEndFunction(k, 0, xPoint, function)
19              return xPoint
20          }
21
22          val aMatrix = create(doubleArrayOf(), numRows = 3, numCols = 2)
23
24          if (k == 0) {
25              for (i in 0 until aMatrix.numRows()) {
26                  val derivValues = derivCondFunction(xPoint)[i]
27                  for (j in 0 until aMatrix.numCols()) {
28                      aMatrix[i, j] = derivValues[j]
29                  }
30              }
31
32
33              val step = mat[0.0, 0.0, 0.0]
34              for (i in 0 until 3) {
35                  step[i] = -condFunctions(xPoint)[i]
36              }
37
38              val at = aMatrix.transpose()
39              val a_at = aMatrix * at
40              val at_inv = create(doubleArrayOf(), numRows = 3, numCols = 3)
41
42              at_atinv = at * at_inv
43              deltaX = at_atinv * step
44              deltaXNorm = deltaX.normF()
45          } else {
46              deltaX = create(doubleArrayOf(), numCols = 2, numRows = 1)
47              deltaXNorm = 0.0
48          }
49
50          val currentGrad = create(derivFunction(xPoint).toDoubleArray())
51          val at_atinv_a = at_atinv * aMatrix
52
53          val newDeltaX = - (eye(2) - at_atinv_a) * currentGrad
54          val newDeltaXNorm = newDeltaX.normF()
55
56          val condition1 = if (k == 0) {
```

```
57              deltaXNorm <= eps
58          } else {
59              true
60          }
61          val condition2 = newDeltaXNorm <= eps
62
63          if (condition1 && condition2) {
64              PrintUtils.printInfoEndFunction(k, 0, xPoint, function)
65              return xPoint
66          }
67
68          fun gradientFun(x: Double) = function(xPoint + x * newDeltaX)
69
70          val minimum = bisectionMethod(eps, Interval(0.0, 2.0), ::gradientFun)
71          xPoint += minimum * newDeltaX + deltaX
72      } while (true)
73  }
```

5 Результаты.

При последовательном запуске всех алгоритмов со следующими параметрами -

$$epsilon = 10^{-4} \tag{11}$$

были получены следующие результаты:

Листинг 5. Результаты выполнения программ.

```
Start Penalty Function:
         Iteration(s): 2
         f(mat[ 0.98940341787353,  0.97886411115369 ]) = 10.000224726422337

Start Barrier Function:
         Iteration(s): 10
         f(mat[ 1.00142727827899,  1.00285841135583 ]) = 10.000004074411768

Start Lagrange  Functions:
         Iteration(s): 1
         f(mat[ 0.91982772712493,  0.8456562435792 ]) = 10.012864294758998

Start Gradient Projections:
         Iteration(s): 355
         f(mat[ 1.00000522250428,  1.00000692408599 ]) = 10.000000000674403
```

Все результаты с небольшой погрешностью совпадают с результатами полученными с помощью сервиса WolframAlpha.com в пункте 3.