Белорусский государственный университет
Факультет прикладной математики и информатики

Лабораторная работа №3

Метод наименьших квадратов
Построение интерполяционного многочлена Ньютона
Минимизация остатка интерполирования

Вариант №7

Выполнил:
Студент 2 курса 7 группы ФПМИ
Лубенько Алексей Анатольевич

Преподаватель:
Будник Анатолий Михайлович

Минск, 2022

Постановка задачи

Рассмотрим набор различных точек на отрезке $[a, b]$ и обозначим их $x_0 < x_1 < \ldots < x_n \in [a, b]$. В этих точках задано $f(x_i) = f_i,\ i = \overline{0 \ldots n}$.
Требуется восстановить значение $f(x)$ в других точках отрезка $[a, b]$.

$[a, b] = [0.45, 1.45]$
$f(x) = 0.45e^{-x} + 0.55sinx$
$x^* = \frac{31}{60} = 0.51(6), x^{**} = 1, x^{***} = \frac{17}{12} = 1.41(6)$

Методом наименьших квадратов построить аппроксимирующий многочлен степени $m = 5$ предполагая $p(x) \equiv 1$. Вычислить приближения $f(x)$ в x^*, x^{**}, x^{***}. Оценить погрешность.

Значения в таблицу внесены с точностью 10^{-6}. В программе значения считаются с точностью машинного эпсилона

i	0	1	2	3	4
x_i	0.45	0.55	0.65	0.75	0.85
f_i	0.501113	0.547105	0.567773	0.587466	0.605541

i	5	6	7	8	9	10
x_i	0.95	1.05	1.15	1.25	1.35	1.45
f_i	0.621412	0.634555	0.644507	0.650869	0.653306	0.651549

Метод наименьших квадратов

Аппроксимирующий многочлен будем искать в виде:

$$\Phi(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 + c_5 x^5$$

Скалярное произведение:

$$(f, g) = \sum_{i=0}^{10} f_i g_i$$

Коэффициенты многочлена будем находить методом Гаусса из СЛАУ вида:

$$\sum_{i=0}^{5} c_i \left(\sum_{j=0}^{10} x_j^{i+k} \right) = \sum_{j=0}^{10} f_i x_i^k, \quad k = \overline{0 \ldots 5}$$

Погрешность находим по формуле:

$$\Delta f = \sqrt{\sum_{i=0}^{10} (f(x_i) - \Phi(x_i))^2}$$

И будем искать истинную погрешность в точках x^*, x^{**}, x^{***} по формуле $r(x) = f(x) - \Phi(x)$

Листинг программы

```
import kotlin.math.*
object Main {
    private var start = 0.45
    private var step = 0.1
    private var amount_of_segments = 10
    private var approx_polynomial: DoubleArray? = DoubleArray(6)
    private var point_of_build = ArrayOf(0.1 * 2 / 3 + 0.45, 1.0, 1.45 - 0.1 / 3)
    private fun f(x: Double) = 0.45 * exp(-x) + 0.55* sin(x)

    private fun iScaleMultiJ(i: Int, j: Int): Double {
        var sum = 0.0
        for (k in 0..amount_of_segments) {
            sum += (start + k * step).pow((i + j).toDouble())
        }
        return sum
    }

    private fun fScaleMultiI(i: Int): Double {
        var sum = 0.0
        for (k in 0..amount_of_segments) {
            sum += f(start + k * step) * (start + k * step).pow(i.toDouble())
        }
        return sum
    }

    private fun pol(x: Double): Double {
        var sum = approx_polynomial!![approx_polynomial!!.size - 1]
        for (i in 4 downTo 0) {
            sum = sum * x + approx_polynomial!![i]
        }
        return sum
    }

    private fun diverge(): Double {
        val sum = (0..amount_of_segments)
            .map { start + it * step }
            .sumOf { (f(it) - pol(it)).pow(2.0) }
        return sqrt(sum)
    }

    private fun gauss(matrix: Array<DoubleArray>, b: DoubleArray): DoubleArray {
        val size = matrix.size
        for (p in 0 until size) {
            var max = p
            for (i in p + 1 until size) {
                if (abs(matrix[i][p]) > abs(matrix[max][p])) {
                    max = i
                }
            }
            val temp = matrix[p]
            matrix[p] = matrix[max]
            matrix[max] = temp
            val t = b[p]
```

```
            b[p] = b[max]
            b[max] = t
            for (i in p + 1 until size) {
                val alpha = matrix[i][p] / matrix[p][p]
                b[i] -= alpha * b[p]
                for (j in p until size) {
                    matrix[i][j] -= alpha * matrix[p][j]
                }
            }
        }
        val x = DoubleArray(size)
        for (i in size - 1 downTo 0) {
            var sum = 0.0
            for (j in i + 1 until size) {
                sum += matrix[i][j] * x[j]
            }
            x[i] = (b[i] - sum) / matrix[i][i]
        }
        return x
    }

    @JvmStatic
    fun main(args: Array<String>) {
        val matrix = Array(6) { DoubleArray(6) }
        val stool = DoubleArray(6)
        for (i in 0..5) {
            for (j in 0..5) {
                matrix[i][j] = iScaleMultiJ(i, j)
            }
            stool[i] = fScaleMultiI(i)
        }
        approx_polynomial = gauss(matrix, stool)
        println("Коэффициенты многочлена: ")
        for (k in approx_polynomial!!) {
            print("$k ")
        }
        println("\n\nЗначения многочлена в точках восстановления: ")
        for (i in 0..2) {
            println(points_of_rebuilding[i].toString() + " : " +
pol(points_of_rebuilding[i]))
        }
        println("\nПогрешность приближения: " + diverge())
        println("\nИстинная погрешность в указанных точках: ")
        for (i in 0..2) {
            println(points_of_rebuilding[i].toString() + " : " +
(f(points_of_rebuilding[i]) - pol(
                points_of_rebuilding[i])))
        }
    }
}
```

Вывод программы

Коэффициенты многочлена:
0.45012527942074637; 0.09912778173082121; 0.2273341715104189;
-0.1695822161229186; 0.02017603152194313; 0.0011736039155255864

Значения многочлена в точках x^*, x^{**}, x^{***}:
0.5166666666666667 : 0.5401188345955142
1.0 : 0.6283546519765366
1.4166666666666665 : 0.6526142970998626

Погрешность приближения: 4.3090421680747044E-7

Истинная погрешность в указанных точках:
0.5166666666666667 : 2.394547059525465E-7
1.0 : 1.3819495559008743E-7
1.4166666666666665 : 2.223255555211523E-7

Интерполяционный многочлен Ньютона

Многочлен Ньютона имеет вид: $P_n(x) = f(x_0) + (x - x_0)f(x_0, x_1) + \ldots + (x - x_0)\ldots(x - x_{n-1})f(x_0, x_1, \ldots, x_n)$

где

$$f(x_0, x_1, \ldots, x_{k+1}) = \frac{f(x_1, x_2, \ldots, x_{k+1}) - f(x_0, x_1, \ldots, x_k)}{x_{k+1} - x_0}$$

$$f(x_i, x_j) = \frac{f(x_i) - f(x_j)}{x_i - x_j}$$

Истинную погрешность в точках x^*, x^{**}, x^{***} будем искать по формуле $r(x) = f(x) - P_n(x)$

Остаток интерполирования — по формуле

$$r_n(x^*) = w_{n+1}(x^*)f(x^*, x_0, x_1, \ldots, x_n)$$

Аналогичные формулы для x^{**}, x^{***}.

$$w_{n+1}(x) = (x - x_0)(x - x_1)\ldots(x - x_n)$$

Таблица разделенных разностей:

0.45	0.5261	0.2091	-0.0136	-0.1167	0.0236	0.0016	-2.2E-4	-1.1E-4	1.5E-5	4.3E-7	-7.4E-8
0.55	0.5471	0.2067	-0.0487	-0.1073	0.0244	0.0014	-3.0E-4	-1.0E-4	1.6E-5	3.5E-7	
0.65	0.5677	0.1969	-0.0809	-0.0975	0.0252	0.0013	-3.7E-4	-9.1E-5	1.8E-5		
0.75	0.5874	0.1807	-0.1101	-0.0874	0.0258	0.0010	-4.4E-4	-7.9E-5			
0.85	0.6055	0.1581	-0.1364	-0.0770	0.0264	8.2E-4	-4.9E-4				
0.95	0.6214	0.1314	-0.1595	-0.0665	0.0268	5.2E-4					
1.05	0.6345	0.0995	-0.1794	-0.0557	0.0271						
1.15	0.6445	0.0636	-0.1962	-0.0449							
1.25	0.6508	0.0243	-0.2097								
1.35	0.6533	-0.0175									
1.45	0.6515										

В таблице k столбцу соответствует разделенная разность $k - 1$ порядка. Значения в таблицу внесены с четырьмя знаками после запятой. В программе разделенные разности считаются с точностью машинного эпсилона.

Листинг программы

```
object Main {
    private var start = 0.45
    private var step = 0.1
    private var value = DoubleArray(11)
    private var newtonValues = DoubleArray(3)
    var point_of_build = doubleArrayOf(0.1 * 2 / 3 + 0.45, 1.0, 1.45 - 0.1 / 3)
    private var diffMatr = Array(11) { DoubleArray(11) }
    private fun calculateAllValues() {
        for (i in 0..10) {
            value[i] = start + i * step
            diffMatr[i][i] = f(value[i])}
        for (d in 1..10) {
            var i = 0
            while (i + d <= 10) {
                diffMatr[i][i + d] = (diffMatr[i + 1][i + d] - diffMatr[i][i + d -
1]) / (value[i + d] - value[i])
                i++}}}
    private fun f(x: Double): Double = 0.45 * exp(-x) + 0.55 * sin(x)
    private fun valueOfPolynomOfNewton(x: Double): Double {
        var sum = f(value[0])
        var mult = 1.0
        (0..9).forEach { i ->
            mult *= x - value[i]
            sum += mult * diffMatr[0][i + 1]
        }
        return sum
    }
    private fun remainder(x: Double): Double {
        var r = valueOfPolynomOfNewton(x)
        for (i in 0..10) r = (r - diffMatr[0][i]) / (x - value[i])
        for (i in 0..10) r *= x - value[i]
        return abs(r)
    }
    @JvmStatic
    fun main(args: Array<String>) {
        calculateAllValues()
        for (i in 0..2) newtonValues[i] =
valueOfPolynomOfNewton(points_of_rebuilding[i])
        println("Приближенные значения функции в точках восстановления:")
        for (i in 0..2) println(points_of_rebuilding[i].toString() + " : " +
newtonValues[i])
        println("Погрешности функции в точках восстановления:")
        for (i in 0..2)
            println(points_of_rebuilding[i].toString() + " : " +
(f(points_of_rebuilding[i]) - newtonValues[i]))
        println("Остатки интерполирования в точках восстановления:")
        for (i in 0..2) println(points_of_rebuilding[i].toString() + " : " +
remainder(points_of_rebuilding[i]))
    }
```

Вывод программы

```
Приближенные значения функции в точках восстановления:
0.5166666666666667 : 0.540119074050245
1.0 : 0.6283547901714914
1.4166666666666665 : 0.6526145194253701

Истинные погрешности функции в точках восстановления:
0.5166666666666667 : -2.475797344914099E-14
1.0 : 7.771561172376096E-14
1.4166666666666665 : 4.807265696626928E-14

Остатки интерполирования в точках восстановления:
0.5166666666666667 : 1.78176691299843E-13
1.0 : 1.6271077885158906E-13
1.4166666666666665 : 7.618590589649444E-13
```

Минимизация остатка интерполирования

Для минимизации остатка выберем оптимальные узлы интерполирования по формуле:

$$x_k = \frac{a+b}{2} + \frac{b-a}{2}\cos\frac{(2k+1)\pi}{2(n+1)}\ k = \overline{0,n}$$

По посчитаннным узлам построим многочлен Ньютона. Вид многочлена и способ построения указан выше.

Значения в таблицу внесены с точностью 10^{-6} и перенумерованы. В программе значения считаются с точностью машинного эпсилона

i	0	1	2	3	4
x_i	0.455089	0.495184	0.572125	0.679679	0.809133
f_i	0.527224	0.535612	0.551727	0.573749	0.598388

i	5	6	7	8	9	10
x_i	0.950000	1.090866	1.220320	1.327874	1.404815	1.444910
f_i	0.621411	0.634555	0.639031	0.653119	0.652876	0.651742

Погрешность на всем отрезке вычислим по формуле

$$|r_n(x)| \le \frac{\max\limits_{[a,b]}\left|f^{(n+1)}(x)\right|}{(n+1)!} * \frac{(b-a)^{n+1}}{2^{2n+1}}\text{, где } n = 10$$

$$f^{(11)}(x) = -0.45e^{-x} - 0.55cosx$$

$$\max_{[0.45,1.45]}\left|f^{(11)}(x)\right| = 0.782$$

$$|r_n(x)| \le \frac{0.782}{11! * 2^{21}} = 9.341 * 10^{-15}$$

Листинг программы

```
object Main {
    private var start = 0.45
    private var finish = 1.45
    private var value = DoubleArray(11)
    private var newtonValues = DoubleArray(3)
    var points_of_build = doubleArrayOf(0.1 * 2 / 3 + 0.45, 1.0, 1.45 - 0.1 / 3)
    private var diffMatr = Array(11) { DoubleArray(11) }
    private fun calculateAllValues() {
        for (i in 0..10) {
            value[i] = (start + finish) / 2 + (finish - start) / 2 * cos((2 * i +
1) * PI / 2 / (10 + 1))
            diffMatr[i][i] = f(value[i])
        }
        for (d in 1..10) {
            var i = 0
            while (i + d <= 10) {
                diffMatr[i][i + d] = (diffMatr[i + 1][i + d] - diffMatr[i][i + d -
1]) / (value[i + d] - value[i])
                i++
            }
        }
    }
    private fun f(x: Double): Double = 0.45 * exp(-x) + 0.55 * sin(x)
    private fun valueOfPolynomOfNewton(x: Double): Double {
        var sum = f(value[0])
        var mult = 1.0
        (0..9).forEach { i ->
            mult *= x - value[i]
            sum += mult * diffMatr[0][i + 1]
        }
        return sum
    }
    private fun remainder(x: Double): Double {
        var r = valueOfPolynomOfNewton(x)
        for (i in 0..10) r = (r - diffMatr[0][i]) / (x - value[i])
        for (i in 0..10) r *= x - value[i]
        return abs(r)
    }
    @JvmStatic
    fun main(args: Array<String>) {
        calculateAllValues()
        for (i in 0..2) newtonValues[i] =
valueOfPolynomOfNewton(points_of_rebuilding[i])
        println("Приближенные значения функции в точках восстановления:")
        for (i in 0..2) println(points_of_rebuilding[i].toString() + " : " +
newtonValues[i])
        println("Погрешности функции в точках восстановления:")
        for (i in 0..2)
            println(points_of_rebuilding[i].toString() + " : " +
(f(points_of_rebuilding[i]) - newtonValues[i]))
        println("Остатки интерполирования в точках восстановления:")
        for (i in 0..2) println(points_of_rebuilding[i].toString() + " : " +
remainder(points_of_rebuilding[i]))}}
```

Выходные данные

Приближенные значения функции в точках восстановления:
0.5166666666666667 : 0.5401190740502151
1.0 : 0.6283547901714868
1.4166666666666665 : 0.6526145194254146

Истинные погрешности функции в точках восстановления:
0.5166666666666667 : 5.10702591327572E-16
1.0 : 5.440092820663267E-17
1.4166666666666665 : 3.552713678800501E-16

Остатки интерполирования в точках восстановления:
0.5166666666666667 : 1.56443507842804E-16
1.0 : 1.451594698494982E-16
1.4166666666666665 : 8.455100081128709E-16

Погрешность интерполирования на всем отрезке:
9.341596964557373E-15

Выводы:

Сравним многочлен Ньютона и МНК. В МНК невязка в точках воостановления имела порядок 10^{-7}, что значительно больше полученной невязки, используя многочлен Ньютона. Это может быть связано с тем, что многочлен Ньютона имеет большую степень, чем многочлен, полученный МНК. Построив многочлен Ньютона по узлам Чебышева, мы добились увеличения точности и уменьшения остатков интерполирования. Истинные погрешности меньше, чем отстатки интерполирования, что соответствует теории.