Университет ИТМО, факультет ПИиКТ

Лабораторная работа №5

Дисциплина: Вычислительная математика

Вариант 17

Выполнил: Чайка Алексей

Группа: Р3214

Преподаватель: Малышева Т. А.

Санкт-Петербург, 2020 год

**Текст задания:**

**Лабораторная работа 5. «Интерполяция функции»**

**Исходные данные:**

|  |  |
| --- | --- |
| **Таблица 1** | |
|  | |
| **x** | **y** |
| 0,298 | 3,2557 |
| 0,303 | 3,1764 |
| 0,310 | 3,1218 |
| 0,317 | 3,0482 |
| 0,323 | 2,9875 |
| 0,330 | 2,9195 |
| 0,339 | 2,8359 |

|  |  |
| --- | --- |
| **Таблица 5** | |
|  | |
| **x** | **y** |
| 0,25 | 1,2557 |
| 0,30 | 2,1764 |
| 0,35 | 3,1218 |
| 0,40 | 4,0482 |
| 0,45 | 5,9875 |
| 0,50 | 6,9195 |
| 0,55 | 7,8359 |

*Для исследования использовать:*

* линейную и квадратичную интерполяцию;
* многочлен Лагранжа;
* многочлен Ньютона.

**Методика проведения исследования:**

* С помощью линейной и квадратичной интерполяции найти приближенное значение функции при х= Х1 (см. табл. 1).
* Найти приближенное значение функции при х= Х1 (см. табл. 1) с помощью многочлена Лагранжа.
* Используя первую или вторую интерполяционную формулу Ньютона, вычислить значения функции при данных значениях аргумента (для значения Х2 и Х3, см. табл. 5).
* Вычислить значения функции, используя интерполяционную формулу Ньютона для не равноотстоящих узлов (для х=Х4, см. табл. 1). При вычислениях учитывать только разделенные разности первого и второго порядков. Вычисления произвести дважды, используя различные узлы.

**Программная реализация задачи:**

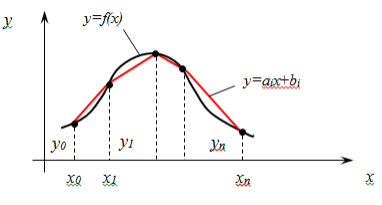
* Предусмотреть ввод исходных данных (исходные таблицы) из файла.
* Предусмотреть ввод значения аргумента, для которого вычисляется приближенное значение функции, с клавиатуры.
* Реализовать численные методы интерполирования, каждый метод в отдельной функции/классе.
* Предусмотреть вывод результатов на экран.

**Цель работы:** решить задачу интерполяции, найти значения функции при заданных значениях аргумента, отличных от узловых точек.

**Ход работы**

**Линейная интерполяция:**

*Идея метода:* простейшим и часто используемым видом локальной интерполяции является линейная интерполяция. Она состоит в том, что заданные точки , соединяются прямолинейными отрезками, и функция приближается к ломаной с вершинами в данных точках.



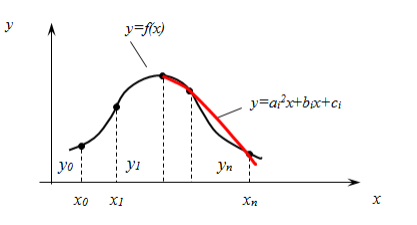
Уравнения каждого отрезка ломаной линии в общем случае разные. Поскольку имеется n интервалов , то для каждого из них в качестве уравнения интерполяционного полинома используется уравнение прямой, проходящей через две точки. В частности, для — го интервала можно написать уравнение прямой, проходящей через точки и в виде:

Отсюда:

Следовательно, при использовании линейной интерполяции сначала нужно определить интервал, в который попадает значение аргумента x, а затем подставить его в формулу (2) и найти приближенное значение функций в этой точке.

**Квадратичная интерполяция:**

*Идея метода:* в случае квадратичной интерполяции в качестве интерполяционной функции на отрезке принимается квадратный трехчлен. Уравнения квадратного трехчлена имеет вид:



Для определения неизвестных коэффициента необходимы три уравнения. Ими служат условия прохождения параболы (3) через три точкиЭти условия можно записать в виде:

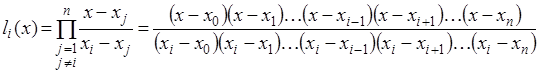
Интерполяция для любой точки проводится по трем ближайшим ней узлам.

**Многочлен Лагранжа:**

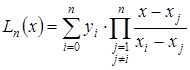
*Идея метода:* при глобальной интерполяции на всем интервале  строится единый многочлен. Одной из форм записи интерполяционного многочлена для глобальной интерполяции является многочлен Лагранжа:



где  – базисные многочлены степени *n*:



То есть многочлен Лагранжа можно записать в виде:



Многочлен удовлетворяет условию

Это условие означает, что многочлен равен нулю при каждом  кроме , то есть

 – корни этого многочлена.

Таким образом, степень многочлена  равна *n* и при  обращаются в ноль все слагаемые суммы, кроме слагаемого с номером , равного .

Выражение применимо как для равноотстоящих, так и для не равноотстоящих узлов. Погрешность интерполяции методом Лагранжа зависит от свойств функции *,* от расположения узлов интерполяции и точки *x*. Полином Лагранжа имеет малую погрешность при небольших значениях *n* (*n*<20). При больших *n* погрешность начинает расти, что свидетельствует о том, что метод Лагранжа не сходится (то есть его погрешность не убывает с ростом *n*).

**Многочлен Ньютона:**

*Идея метода:* другая форма записи интерполяционного многочлена – интерполяционный многочлен Ньютона с разделенными разностями. Пусть функция  задана с произвольным шагом, и точки таблицы значений пронумерованы в произвольном порядке.

**Разделенные разности** нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка определяются через разделенные разности нулевого порядка:



Разделенные разности второго порядка определяются через разделенные разности первого порядка:



Разделенные разности *k*-го порядка определяются через разделенные разности порядка :



Используя понятие разделенной разности интерполяционный многочлен Ньютона можно записать в следующем виде:



**Интерполяционный полином Ньютона для равноотстоящих узлов:**

Узлы интерполирования *x*0, *x*1, ..., *xn* называются равноотстоящими, если:

 , где*h* - шаг интерполирования, *.*

Конечные разности являются рабочим аппаратом при изучении функций, заданных таблицей значений в равноотстоящих узлах.

**Конечными разностями первого порядка** называют величины:

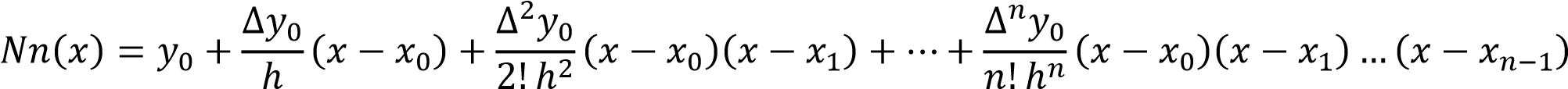
, 

**Конечными разностями второго порядка** называют величины:

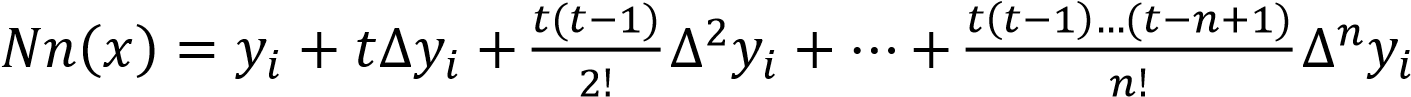


**Конечными разностями k-го порядка** называют величины:





Введем обозначение: ( . Тогда получим формулу Ньютона, которая называется *первой интерполяционной формулой Ньютона для интерполирования вперед*:



Полученное выражение может аппроксимировать функцию на левом половине отрезка.

Для правой половины отрезка разности вычисляют справа налево: (. Тогда получим формулу Ньютона, которая называется **второй интерполяционной формулой Ньютона для интерполирования назад:**



При **экстраполировании** для отыскания значений функции для используется первый интерполяционный многочлен Ньютона. В этом случае *t =< 0* и говорят, что первая интерполяционная формула Ньютона применяется для **экстраполирования назад**.

При отыскании значений функции для используется второй интерполяционный многочлен Ньютона.

В этом случае *t => 0* и говорят, что вторая интерполяционная формула Ньютона применяется для **экстраполирования вперед**.

**Вычисления:**

**1.** Найти приближенное значение функции y=f(x) при х=0,321 для таблицы 1.

**Решение:** используем линейную интерполяцию. Значение x=0,321 находится между узлами . Тогда:

**Решение:** используем квадратичную интерполяцию. Составим систему уравнений для ближайших узлов к точке x=0,321: , соответственно .

В результате решения системы получим:

**2.** Найти приближенное значение функции y=f(x) при х=0,321 для таблицы 1 с помощью многочлена Лагранжа.

**Решение:** найдем:

**3.** Используя первую или вторую интерполяционную формулу Ньютона найти приближенное значение функции для , для таблицы 5.

**Решение:** для вычисления значение функции при воспользуемся формулой Ньютона для интерполирования вперед, т.к. лежит в левой половине отрезка.

Поскольку , то при использовании первой интерполяционной формулой Ньютона для интерполирования вперед необходимо вычислить:

Примем . Тогда

Конечные разности первого порядка:

Тогда:

Конечные разности второго порядка:

Тогда:

Конечные разности третьего порядка:

Тогда:

Конечные разности четвертого порядка:

Тогда:

Конечные разности пятого порядка:

Тогда:

Конечные разности шестого порядка:

Тогда:

Получаем:

**Решение:** для вычисления значение функции при воспользуемся формулой Ньютона для интерполирования назад, т.к. лежит в правой половине отрезка.

Тогда

Конечные разности остаются те же, что и в первой формуле.

Получаем:

**4.** Используя интерполяционную формулу Ньютона для неравноотстоящих узлов найти приближенное значение функции для х=0,336 для таблицы 1. При вычислениях учитывать только разделенные разности первого и второго порядков. Вычисления провести дважды, используя различные узлы.

**Решение:** вычисления произведем по формуле:

Для вычисления значения функции при за возьмем сначала 0.317, затем 0.323

Для

Принимаем

**Листинг:**

Main.java

**package** pepe.lmao;  
  
**import** pepe.lmao.interpolation.Interpolation;  
  
**import** java.io.BufferedReader;  
**import** java.io.FileReader;  
**import** java.io.IOException;  
**import** java.util.Scanner;  
  
**public class** Main {  
 **static final double**[] ***x*** = **new double**[7];  
 **static final double**[] ***y*** = **new double**[7];  
 **public static void** main(String[] args) **throws** IOException {  
 Scanner scanner = **new** Scanner(System.***in***);  
 *readFromFile*(**"table\_1.txt"**);  
 System.***out***.println(**"Set argument for Lagrange Interpolation: "**);  
 Interpolation.*solveByLagrange*(***x***,***y***,scanner.nextDouble());  
 System.***out***.println(**"Set argument for Newton Polynomial Interpolation: "**);  
 Interpolation.*solveByNewtonPoly*(***x***,***y***,scanner.nextDouble());  
 *readFromFile*(**"table\_2.txt"**);  
 System.***out***.println(**"Set argument for Newton Interpolation: "**);  
 Interpolation.*solveByNewton*(***x***,***y***,scanner.nextDouble());  
 }  
  
 **public static void** readFromFile(String filename) **throws** IOException {  
 String row;  
 String[] data;  
 BufferedReader reader = **new** BufferedReader(**new** FileReader(filename));  
 **int** i = 0;  
 **while** ((row = reader.readLine()) != **null**) {  
 data = row.split(**" "**);  
 ***x***[i] = Double.*parseDouble*(data[0]);  
 ***y***[i] = Double.*parseDouble*(data[1]);  
 i++;  
 }  
 }  
}

Interpolation.java

**package** pepe.lmao.interpolation;  
  
**public class** Interpolation {  
  
 **public static void** solveByLagrange(**double**[] x, **double**[] y, **double** argument) {  
 LagrangeInterpolation lagrangeInterpolation = **new** LagrangeInterpolation(x, y, argument);  
 lagrangeInterpolation.solve();  
  
 System.***out***.println(**"-->Lagrange Interpolation result: "** + lagrangeInterpolation.getRes());  
 }  
  
 **public static void** solveByNewtonPoly(**double**[] x, **double**[] y, **double** argument) {  
 NewtonPolyInterpolation newtonPolyInterpolation = **new** NewtonPolyInterpolation(x,y,argument);  
 newtonPolyInterpolation.solve();  
  
 System.***out***.println(**"-->Newton Polynomial Interpolation result: "** + newtonPolyInterpolation.getRes());  
 }  
   
 **public static void** solveByNewton(**double**[] x, **double**[] y, **double** argument) {  
 NewtonInterpolation newtonInterpolation = **new** NewtonInterpolation(x,y,argument);  
 newtonInterpolation.solve();  
  
 System.***out***.println(**"-->Newton Interpolation result: "** + newtonInterpolation.getRes());  
 }  
}

LagrangeInterpolation.java

**package** pepe.lmao.interpolation;  
**import** lombok.Data;  
@Data  
**public class** LagrangeInterpolation {  
 **private double**[] **x**;  
 **private double**[] **y**;  
 **private double argument**;  
 **private int n**;  
 **private double res**;  
 **private double l**;  
  
 **public** LagrangeInterpolation(**double**[] x, **double**[] y, **double** argument) {  
 **this**.**x** = x;  
 **this**.**y** = y;  
 **this**.**n** = x.**length**;  
 **this**.**argument** = argument;  
 }  
  
 **public void** solve() {  
 **for** (**int** i = 0; i < **n**; i++) {  
 **double** numerator;  
 **double** denominator = numerator = 1;  
 **for** (**int** j = 0; j < **n**; j++) {  
 **if** (i != j) {  
 numerator \*= (**argument** - **x**[j]);  
 denominator \*= (**x**[i] - **x**[j]);  
 }  
 }  
 **l** += numerator / denominator \* **y**[i];  
 }  
 setRes(**l**);  
 }  
}

NewtonPolyInterpolation.java

**package** pepe.lmao.interpolation;  
  
**import** lombok.Data;  
  
@Data  
**public class** NewtonPolyInterpolation {  
 **private double**[] **x**;  
 **private double**[] **y**;  
 **private double argument**;  
 **private int n**;  
  
 **private double**[] **node\_x**;  
 **private double**[] **node\_y**;  
 **private double res**;  
  
 **public** NewtonPolyInterpolation(**double**[] x, **double**[] y, **double** argument) {  
 **this**.**x** = x;  
 **this**.**y** = y;  
 **this**.**n** = x.**length**;  
 **this**.**argument** = argument;  
 }  
  
 **public void** solve() {  
 **for** (**int** i = 0; i < **n**; i++) {  
 **if** (**argument** < **x**[**n** - 1] && **argument** > **x**[**n** - 1 - 1] || **argument** > **x**[i] && (**n** - 1 - i) < 3) {  
 **for** (**int** j = 1; j >= 0; j--) {  
 **node\_x** = **new double**[]{**x**[**n** - 1 - 2-j], **x**[**n** - 1 - 1-j], **x**[**n** - 1-j]};  
 **node\_y** = **new double**[]{**y**[**n** - 1 - 2-j], **y**[**n** - 1 - 1-j], **y**[**n** - 1-j]};  
 **res** += calc();  
 }  
 **break**;  
 } **else if** (**argument** > **x**[0] && **argument** < **x**[1] || **argument** > **x**[i] && (**n** - 1 - i) >= 3) {  
 **for** (**int** j = 0; j <= 1; j++) {  
 **node\_x** = **new double**[]{**x**[i + j], **x**[i + j + 1], **x**[i + j + 2]};  
 **node\_y** = **new double**[]{**y**[i + j], **y**[i + j + 1], **y**[i + j + 2]};  
 **res** += calc();  
 }  
 **break**;  
 }  
 }  
 setRes(**res**/2);  
 }  
  
 **private double** calc() {  
 **return** f(0) + f(0, 1) \* (**argument** - **node\_x**[0]) + f(0, 1, 2) \* (**argument** - **node\_x**[0]) \* (**argument** - **node\_x**[1]);  
 }  
  
 **private double** f(**int** a) {  
 **return node\_y**[a];  
 }  
  
 **private double** f(**int** a, **int** b) {  
 **return** (f(b) - f(a)) / (**node\_x**[b] - **node\_x**[a]);  
 }  
  
 **private double** f(**int** a, **int** b, **int** c) {  
 **return** (f(b, c) - f(a, b)) / (**node\_x**[c] - **node\_x**[a]);  
 }  
}

NewtonInterpolation.java

**package** pepe.lmao.interpolation;  
  
**import** lombok.Data;  
**import** org.apache.commons.math3.util.FastMath;  
  
@Data  
**public class** NewtonInterpolation {  
 **private double**[] **x**;  
 **private double**[] **y**;  
 **private double argument**;  
 **private int n**;  
 **private double res**;  
 **private double t**;  
 **static int** *k*;  
  
 **public** NewtonInterpolation(**double**[] x, **double**[] y, **double** argument) {  
 **this**.**x** = x;  
 **this**.**y** = y;  
 **this**.**n** = x.**length**;  
 **this**.**argument** = argument;  
 }  
  
 **public void** solve() {  
 **if** (**argument** < **x**[0]) {  
 **t** = (**argument** - **x**[0]) / (**x**[1] - **x**[0]);  
 backward(**t**);  
 } **else if** (**argument** > **x**[**n** - 1]) {  
 **t** = (**argument** - **x**[**n** - 1]) / (**x**[1] - **x**[0]);  
 forward(**t**);  
 } **else if** (**argument** <= **x**[**n** / 2]) {  
 **for** (**int** i = 0; i < **n** / 2 + 1; i++) {  
 **if** (**argument** > **x**[i]) {  
 **t** = (**argument** - **x**[0]) / (**x**[1] - **x**[0]);  
 *k* = i;  
 }  
 }  
 forward(**t**);  
 } **else** {  
  
 **t** = (**argument** - **x**[**n** - 1]) / (**x**[1] - **x**[0]);  
 backward(**t**);  
 }  
 }  
  
 **private void** forward(**double** t) {  
 **double** res = 0;  
 **double** numerator = t;  
 **for** (**int** i = *k*; i < **n** - 1; i++) {  
 **if** (i == *k*) res = **y**[*k*] + t \* delta(1);  
 **if** (i > *k*) {  
 numerator \*= (t - i);  
 res += numerator / fact(i + 1) \* delta(i + 1);  
 }  
 }  
 setRes(res);  
 }  
  
 **private void** backward(**double** t) {  
 **double** res = 0;  
 **double** numerator = t;  
 **for** (**int** i = 0; i < **n** - 1; i++) {  
 **if** (i == 0) res = **y**[**n** - 1] + t \* delta(1, **n** - 2);  
 **if** (i > 0) {  
 numerator \*= (t + i);  
 res += numerator / fact(i + 1) \* delta(i + 1, **n** - i - 2);  
 }  
 }  
 setRes(res);  
 }  
  
  
 **private double** delta(**int** q) {  
 **return** calcCurrentDelta(q)[*k*];  
 }  
  
 **private double** delta(**int** q, **int** p) {  
 **return** calcCurrentDelta(q)[p];  
  
 }  
  
 **private double**[] calcCurrentDelta(**int** q) {  
 **double**[] res = **new double**[**n**];  
 **if** (q == 1) {  
 res = **new double**[**n**];  
 **for** (**int** i = 0; i < **n** - 1; i++) {  
 res[i] = **y**[1 + i] - **y**[i];  
 }  
 } **else** {  
 **for** (**int** i = 0; i < **n** - 1; i++) {  
 **double** a = calcCurrentDelta(q - 1)[i + 1];  
 **double** b = calcCurrentDelta(q - 1)[i];  
 res[i] = a - b;  
 }  
 }  
 **return** res;  
 }  
  
 **int** fact(**int** n) {  
 **if** (n <= 1)  
 **return** 1;  
 **else  
 return** n \* fact(n - 1);  
 }  
}

**Примеры работы программы:**

|  |  |
| --- | --- |
| table\_1.txt  0.298 3.2557 0.303 3.1764 0.310 3.1218 0.317 3.0482 0.323 2.9875 0.330 2.9195 0.339 2.8359 | table\_2.txt  0.25 1.2557 0.30 2.1764 0.35 3.1218 0.40 4.0482 0.45 5.9875 0.50 6.9195 0.55 7.8359 |
| output  Set argument for Lagrange Interpolation:  *0.321*  -->Lagrange Interpolation result: **3.0067235204385696**  Set argument for Newton Polynomial Interpolation:  *0.336*  -->Newton Polynomial Interpolation result: **2.863458333333334**  Set argument for Newton Interpolation:  *0.272*  -->Newton Interpolation result: **1.2642485556591008** | output  Set argument for Lagrange Interpolation:  *0.321*  -->Lagrange Interpolation result: **3.0067235204385696**  Set argument for Newton Polynomial Interpolation:  *0.336*  -->Newton Polynomial Interpolation result: ***2.863458333333334***  Set argument for Newton Interpolation:  0.545  -->Newton Interpolation result: **7.42258991145375** |

**Вывод:** в результате этой гигантской лабораторной работы, помимо того, что я научился интерполировать функции различными методами и реализовал данные методы с помощью языка программирования Java, я настроил Word для написания фантастических объемов формул и вычислений. Лабораторная работа понравилась, поскольку заставила поразмышлять над оптимизацией кода путем интерпретации формул и вычислений в виде рекурсивных функций и замысловатых for’ов.