МИНОБРНАУКИ РОССИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «ВГУ»)**

Факультет прикладной математики, информатики и механики

Кафедра математического обеспечения ЭВМ

**Численные методы решения**

**спектральных задач линейной алгебры**

Отчёт по лабораторной работе

Студент 3 курса                                                 \_\_\_\_\_\_\_ М.О. Курченков

Преподаватель                                                   \_\_\_\_\_\_\_     О.А. Махинова

Воронеж 2023

**Содержание**

[Постановка задачи 3](#_Toc151595658)

[Анализ задачи 4](#_Toc151595659)

Этапы решения [5](#_Toc151595659)

Тестирование алгоритма7

[Вычислительный эксперимент](#_Toc151595660) 12

[Приложение](#_Toc151595664) 17

Function.java17

InterpolatedFunction[.java 1](#_Toc151595666)8

Section[.java 1](#_Toc151595667)9

Polynomial.java20

CubicSpline.java22

EquationSolver.java24

# Постановка задачи

1. Реализовать классы, производящие интерполяцию заданной функции следующими методами: полином Ньютона по коэффициентам b[i], кубические сплайны;
2. Протестировать корректность работы классов;
3. Провести вычислительные эксперименты;
4. Проанализировать полученные результаты.

# Анализ задачи

Входные данные представлены интерполируемой функцией: f(x) ∈ R→R, отрезком разбиения [a, b] ∈ R и n ∈ N — количеством подотрезков разбиения.

Интерполируемая функция представлена абстрактным классом InterpolatedFunction, содержащим поля section (хранит входную информацию об отрезке разбиения и вычисляет при создании различные способы разбиения) и interpolationMethod — способ интерполирования. В наследуемых от абстракции классах CubicSpline и Polynomial присутсвует метод calculateExpression, реализуемый в зависимости от метода интерполирования.

**Этапы решения**

Решение представлено нахождением соответсвующих методу интерполирования коэффициентов:

1. Полином Ньютона требует нахождения коэффициентов b[i] по следующим формулам:

1.1) Формулы нахождения коэффициентов b[k]:

Рис. 1 — общая формула нахождения b[k]

Рис. 2 — формула нахождения коэффициентов b[k] для равномерного разбиения

1.2) Формула вычисления значения полиномиала в точке x:

Рис. 3 — формула значения полинома Ньютона

1. Кубический сплайн требует нахождения матрицы коэффициентов размерности n x 4, где нулевой столбец является «служебным»:

2.1) Коэффициенты вычисляются из следующей системы (при учёте граничных условий ):

Рис. 4 — система уравнений коэффициентов кубического сплайна

2.2) Формула вычисления значения кубического сплайна в точке x:

Рис. 5 — формула значения кубического сплайна

# Тестирование алгоритма

Возьмём тестовую функцию и проверим, действительно ли строится заданная функция по построению графиков. При этом будем на графиках отображать не только узлы интерполирования, но и промежуточные точки для объективности тестирования:

Рис. 6 — тестовая функция

Будем использовать разбиение Чебышева для улучшения точности вычислений, а для метода сплайнов зададим точные граничные условия:

Рис. 7 — производные тестовой функции

Так, графики принимают следующий вид:

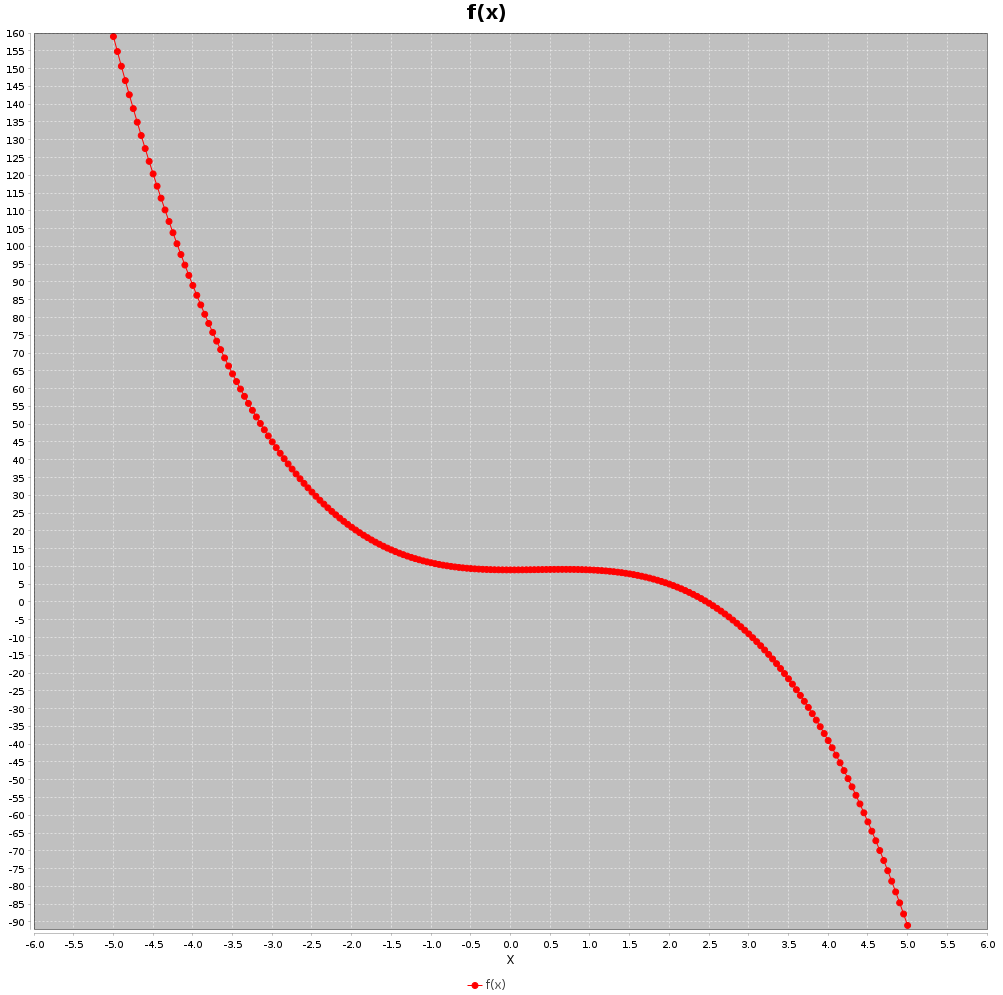


Рис. 8 — график тестовой функции

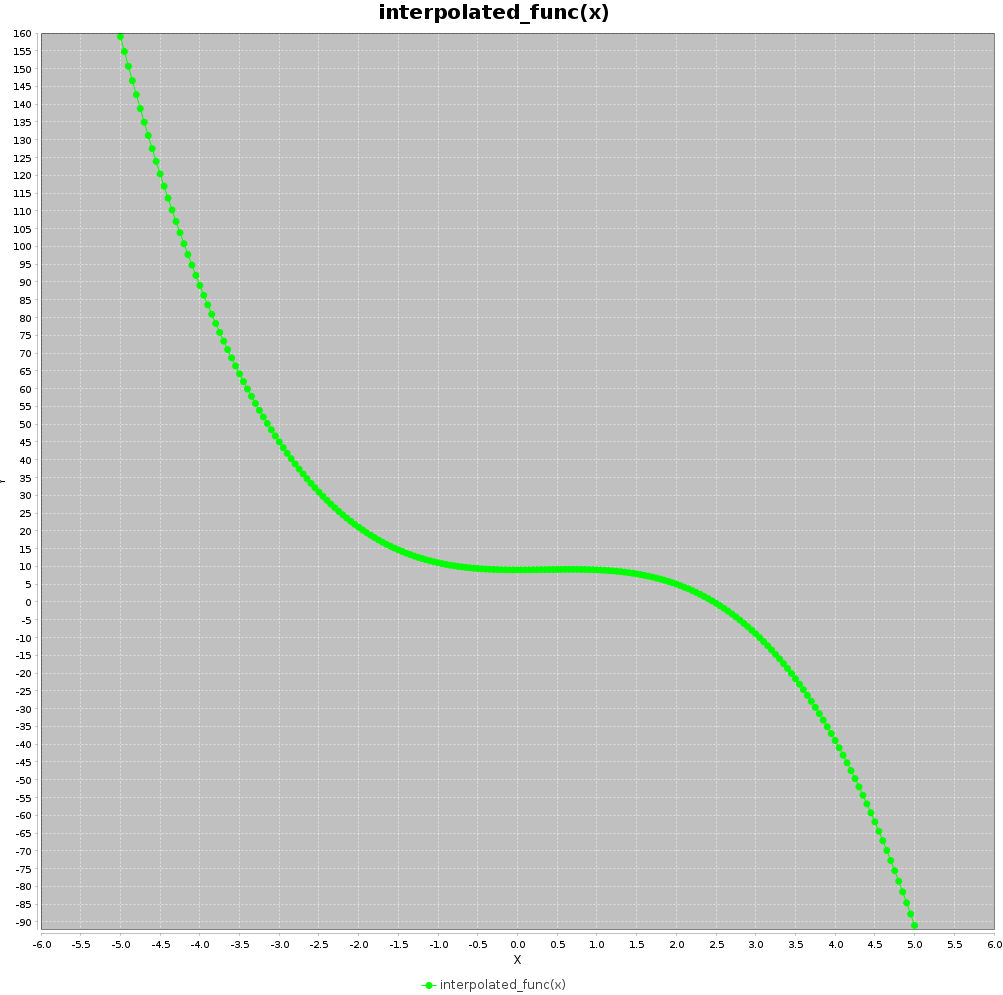
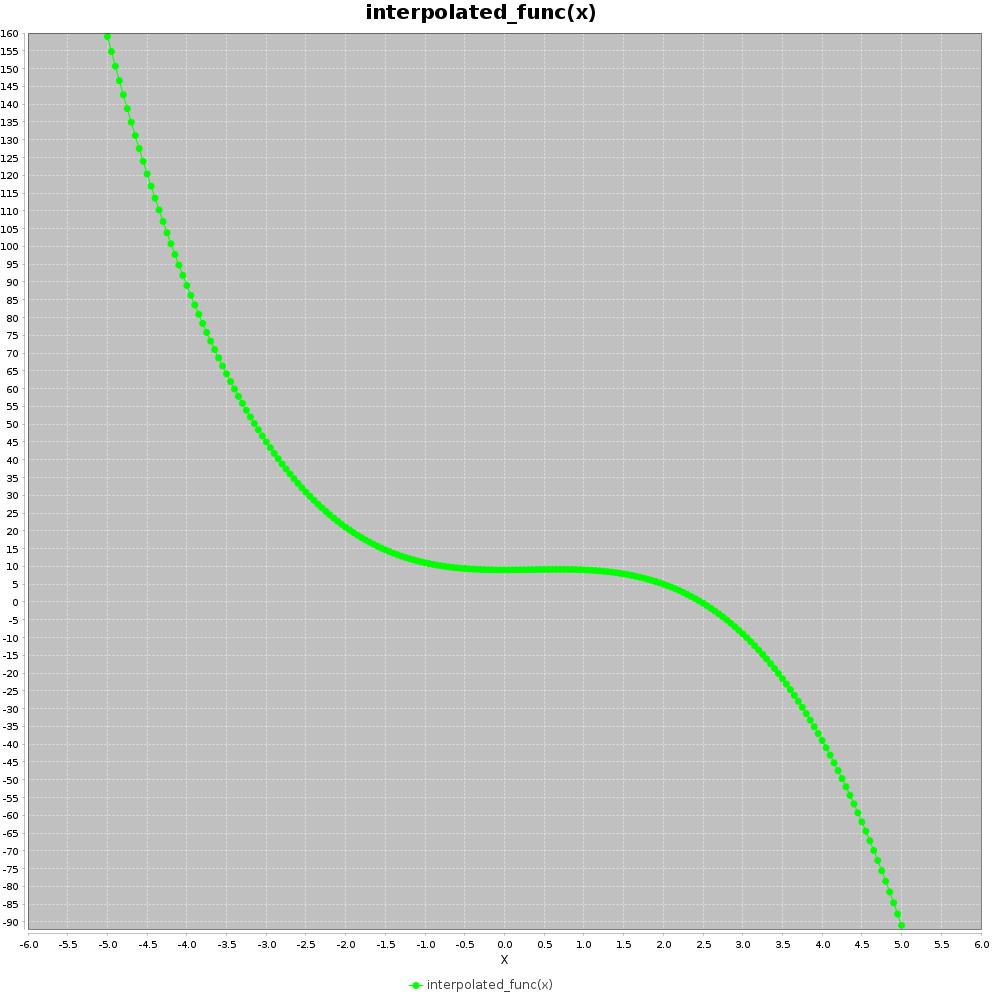


Рис. 9 — график интерполирования кубическими сплайнами

Рис. 10 — график интерполирования полиномом Ньютона

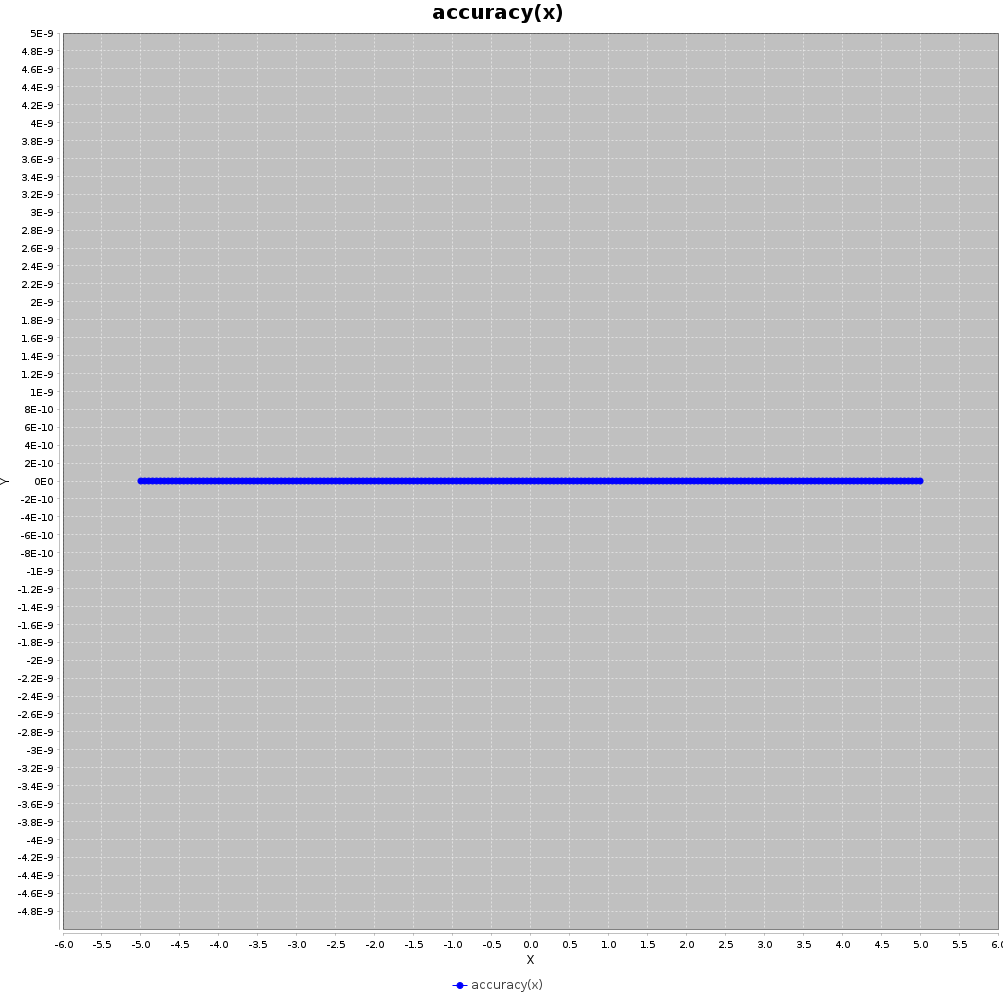
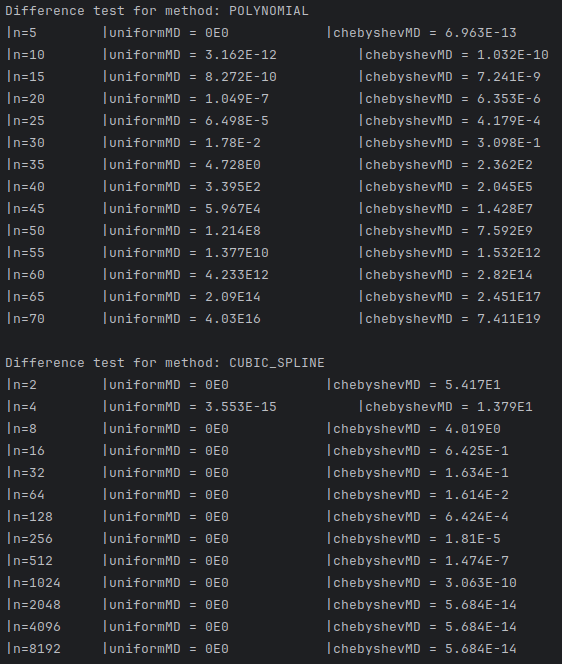


Рис. 11 — график погрешности в обоих случаях

Как видно из графиков, погрешность в обоих случаях стремится к нулю, так как подобраны корректные входные данные, а написанные классы содержат правильную реализацию методов интерполирования.

В дополнение к этому выведем таблицу погрешностей, меняя параметр n: количество отрезков интерполирования в методе кубических сплайнов и степень полинома в методе Ньютона:

Рис. 12 — таблица погрешностей тестовой функции

Обратим внимание, что в случае с тестовой функцией необходимо подобрать верный входной параметр n для обоих методов интерполирования. Вернёмся к этому в рамках вычислительного эксперимента.

# Вычислительный эксперимент

В ходе вычислительного эксперимента будем интерполировать разные функции и сверяться с таблицей погрешностей, указанной выше. Обратимся к ней и проанализируем результаты.

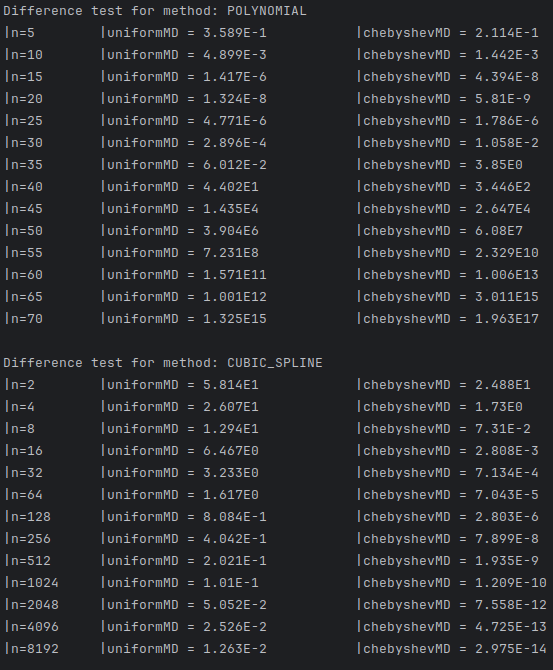
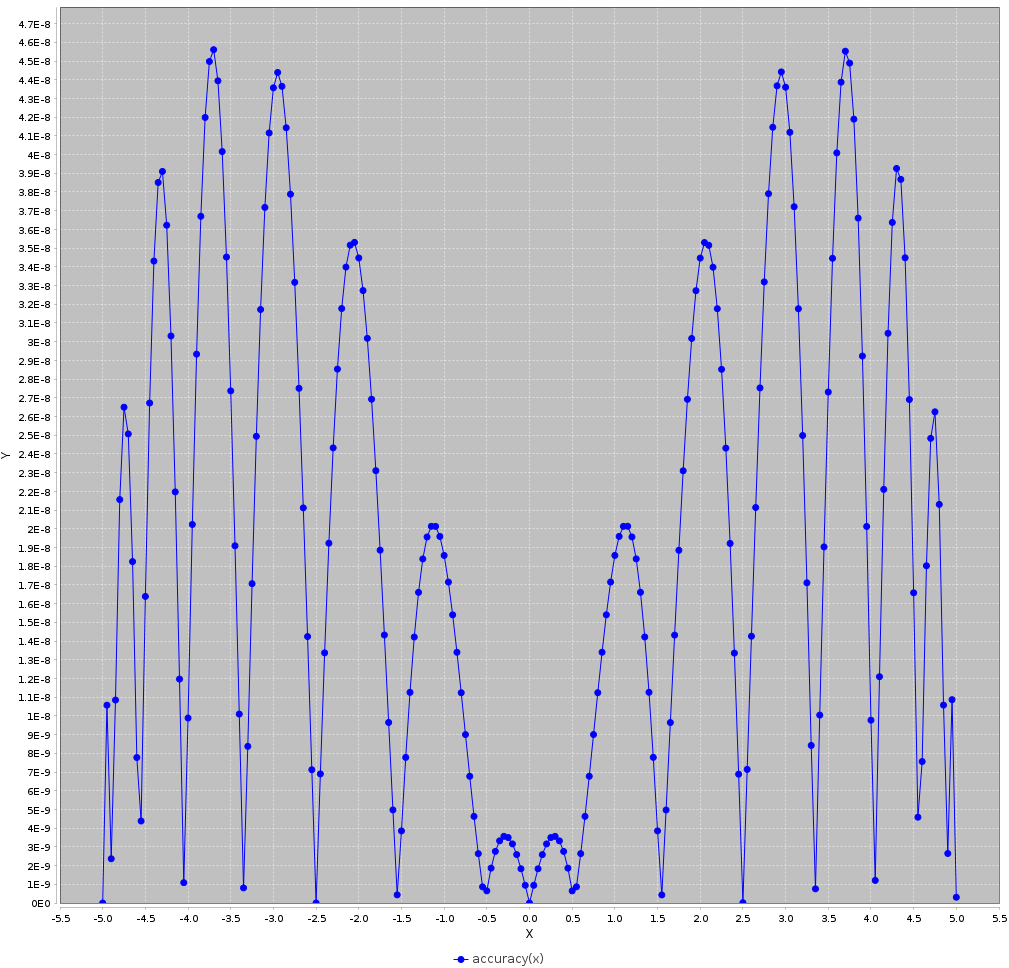
Во первых, обратим внимание на вид функции: она представлена степенной зависимостью третьей степени. Именно поэтому она является примером функции, которая при интерполировании кубическими сплайнами полностью совпадает с исходной. Тем не менее, при увеличении степени полинома метод Ньютона даёт всё большую погрешность. Возьмём контрпример: функцию

Рис. 13 — таблица погрешностей для синуса

Получим, что, в отличие от теоретической сходимости, численный метод интерполирования Ньютона имеет один минимум функции погрешности. Это связано с машинной погрешностью. Однако метод интерполяции кубическими сплайнами даёт сходимость при увеличении количества отрезков разбиения и известных точек. Отметим, что эта зависимость сохраняется для практически любой функции.

Зафиксируем n в значении 15 и выведем график полинома Ньютона:

Рис. 14 — график погрешности для синуса (метод Ньютона)

Обратим внимание на поведение графика погрешности. Так, имеем, что разбиение Чебышева удаляет точки от центра отрезка разбиения. При этом скачки погрешности в этой окрестности уменьшаются. При достижении оптимального значения n интерполяция обеспечивает достаточно низкую погрешность, что видно на рис. 13.

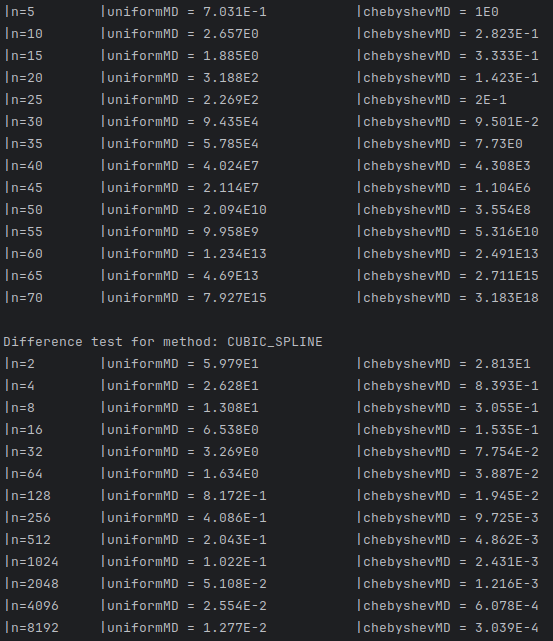
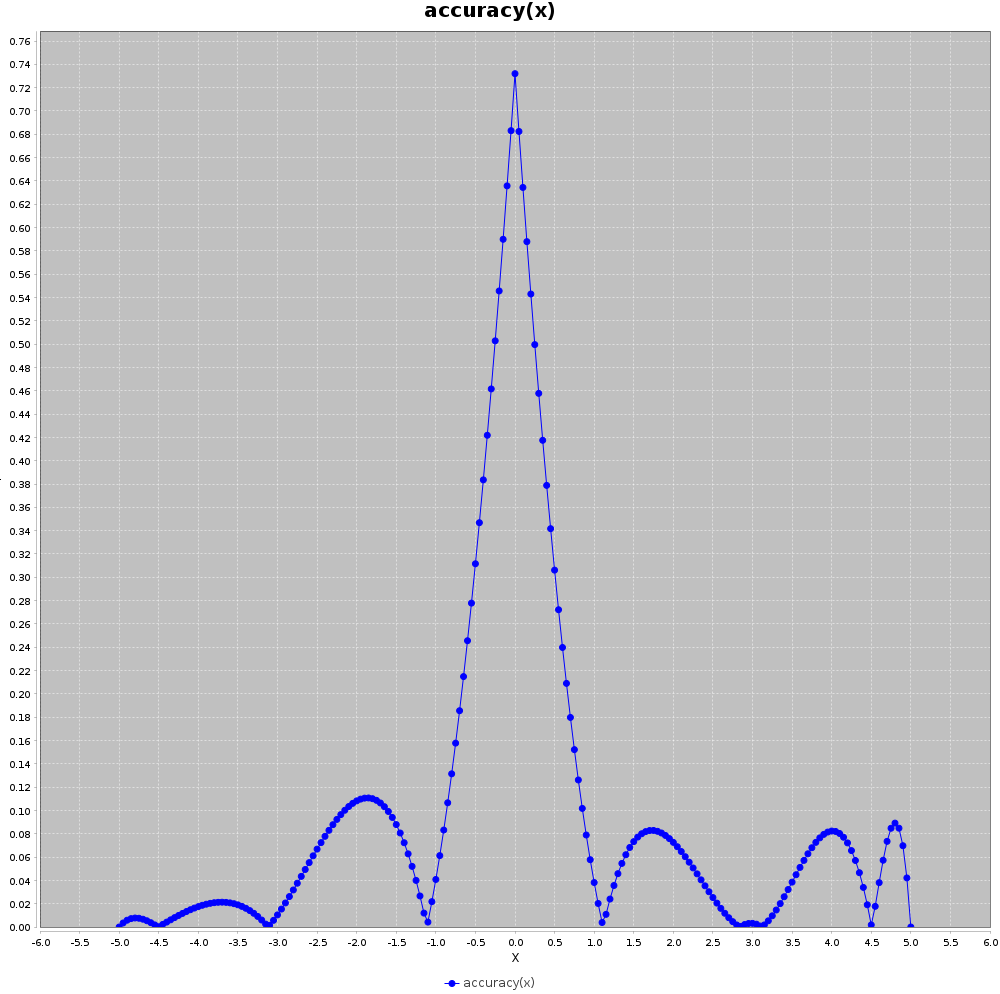
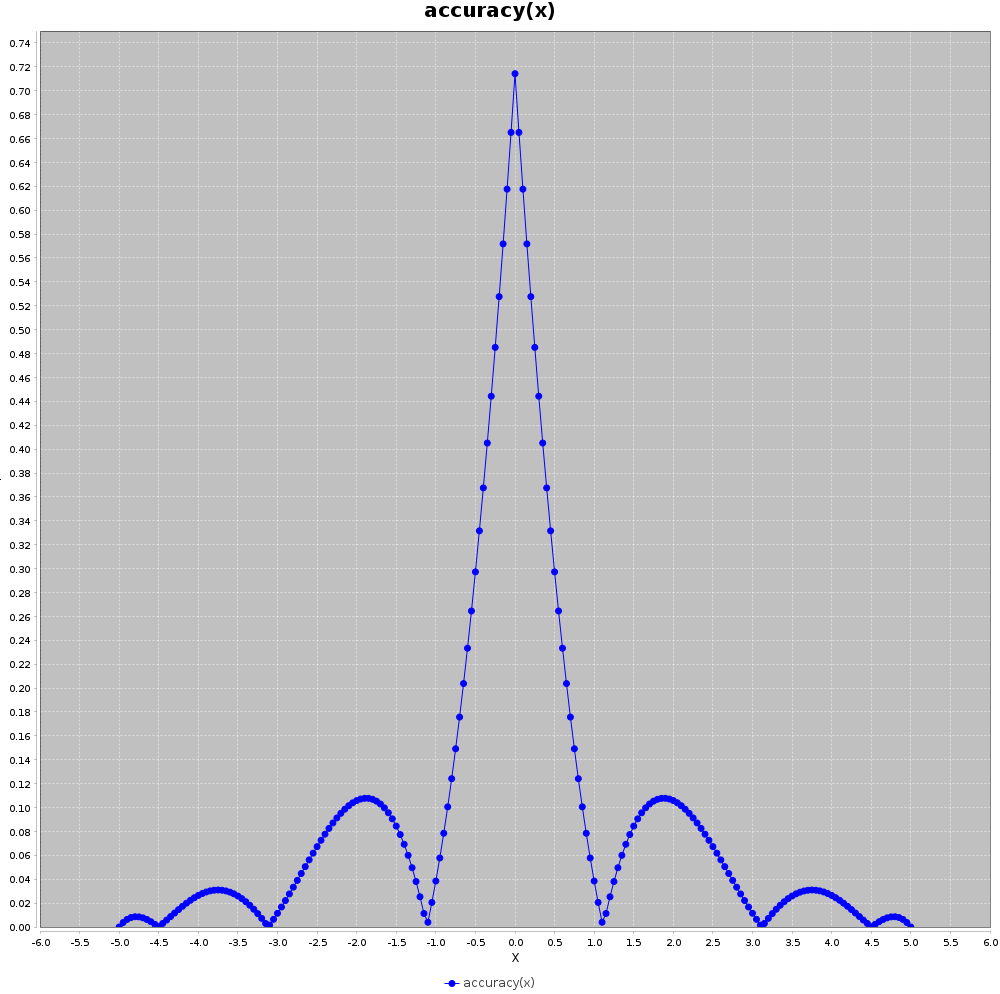
Разберём ещё одну функцию Выведем для неё таблицу погрешностей:

Рис. 15 — таблица погрешностей для модуля

Обратим внимание, что метод интерполирования полиномом Ньютона даже при маленькой степени даёт высокую погрешность, а при увеличении степени расходится с высокой скоростью. При этом кубические спайны гарантируют относительно невысокую погрешность, однако она не сходится в бесконечно малые величины. Это объясняется точкой 0, в которой не существует производной, однако и сплайны, и полином Ньютона гарантируют непрерывность до третьей производной включительно. Подтвердим гипотезу с помощью графиков (для наглядности используем относительно небольшие значения n в обоих случаях):

Рис. 16 — график погрешности для модуля (кубические сплайны)

Рис. 17 — график погрешности для модуля (полином Ньютона)

Из приведённых графиков погрешности отчётливо видно, как точка 0 влияет на погрешность: в ней наблюдается наибольший её скачок. Именно негладкость функции гарантирует несходимость погрешности в бесконечно малые значения в обоих случаях.

# **Приложение**

**Function,java**

package lab.numeric.methods.core.models;  
  
public interface Function {  
 double calculateExpression(double x);  
}

**InterpolatedFunction.****java**

package lab.numeric.methods.core.models;  
  
import lab.numeric.methods.core.enums.InterpolationMethod;  
import lombok.Getter;  
  
@Getter  
public abstract class InterpolatedFunction implements Function {  
  
 private final Section section;  
  
 private final InterpolationMethod interpolationMethod;  
  
 protected InterpolatedFunction(  
 Section section,  
 InterpolationMethod interpolationMethod  
 ) {  
 this.interpolationMethod = interpolationMethod;  
 this.section = section;  
 }  
}

**Section****.****java**

package lab.numeric.methods.core.models;  
  
import lab.numeric.methods.core.enums.SeparationType;  
import lombok.Getter;  
  
@Getter  
public class Section {  
  
 private final double a;  
 private final double b;  
 private final int n;  
 private final double[] separation;  
 private final SeparationType separationType;  
  
 public Section(double a, double b, int n, SeparationType separationType) {  
 this.a = a;  
 this.b = b;  
 this.n = n;  
 this.separationType = separationType;  
 separation = new double[n + 1];  
 if (separationType.equals(SeparationType.*UNIFORM*)) {  
 uniformSeparation();  
 } else {  
 chebyshevSeparation();  
 }  
 }  
  
 private void uniformSeparation() {  
 var h = (b - a) / n;  
 for (int i = 0; i <= n; i++) {  
 separation[i] = a + i \* h;  
 }  
 }  
  
 private void chebyshevSeparation() {  
 for (int i = 0; i <= n; i++) {  
 separation[i] = 0.5 \* (a + b) + (b - a) \* 0.5 \* Math.*cos*(Math.*PI* \* (n - i) / n);  
 }  
 }  
}

**Polynomial****.****java**

package lab.numeric.methods.core.models.impl;  
  
import lab.numeric.methods.core.models.Section;  
import lab.numeric.methods.core.models.Function;  
import lab.numeric.methods.core.enums.InterpolationMethod;  
import lab.numeric.methods.core.enums.SeparationType;  
import lab.numeric.methods.core.models.InterpolatedFunction;  
  
import java.math.BigInteger;  
  
public class Polynomial extends InterpolatedFunction {  
 private final double[] fValues;  
 private final double[] args;  
 private double[] b;  
  
 public Polynomial(Function function, Section section) {  
 super(section, InterpolationMethod.*POLYNOMIAL*);  
 args = section.getSeparation();  
 fValues = new double[args.length];  
 for (int j = 0; j < args.length; j++) {  
 fValues[j] = function.calculateExpression(args[j]);  
 }  
 if (section.getSeparationType().equals(SeparationType.*UNIFORM*)) {  
 uniformSplitDifference();  
 } else {  
 chebyshevSplitDifference();  
 }  
 }  
  
 @Override  
 public double calculateExpression(double x) {  
 double polynom = 0;  
 double expression;  
 for (int i = 0; i < b.length; i++) {  
 expression = b[i];  
 for (int j = 0; j < i; j++) {  
 expression \*= x - args[j];  
 }  
 polynom += expression;  
 }  
 return polynom;  
 }  
  
 private void chebyshevSplitDifference() {  
 b = new double[args.length];  
 for (int i = 0; i < args.length; i++) {  
 for (int j = 0; j <= i; j++) {  
 double denominator = 1;  
 for (int k = 0; k <= i; k++) {  
 if (k != j) {  
 denominator = denominator \* (args[j] - args[k]);  
 }  
 }  
 b[i] += fValues[j] / denominator;  
 }  
 }  
 }  
  
 private void uniformSplitDifference() {  
 b = new double[args.length];  
 var h = args[1] - args[0];  
 for (int k = 0; k < args.length; k++) {  
 for (int i = 0; i <= k; i++) {  
 // i! \* (k - i)! \* h^k  
 double denominator = (  
 factorial(  
 BigInteger.*valueOf*(i)  
 ).multiply(  
 factorial(  
 BigInteger.*valueOf*(k).subtract(BigInteger.*valueOf*(i))  
 )  
 )  
 ).doubleValue() \* (Math.*pow*(h, k));  
 b[k] += fValues[i] \* Math.*pow*(-1, k - i) / denominator;  
 }  
 }  
 }  
  
 private BigInteger factorial(BigInteger x) {  
 if (x.equals(BigInteger.*ZERO*) || x.equals(BigInteger.*ONE*)) {  
 return BigInteger.*ONE*;  
 } else {  
 return x.multiply(factorial(x.subtract(BigInteger.*ONE*)));  
 }  
 }  
}

**CubicSpline****.****java**

package lab.numeric.methods.core.models.impl;  
  
import lab.numeric.methods.EquationSolver;  
import lab.numeric.methods.core.models.Section;  
import lab.numeric.methods.core.models.Function;  
import lab.numeric.methods.core.enums.InterpolationMethod;  
import lab.numeric.methods.core.models.InterpolatedFunction;  
  
public class CubicSpline extends InterpolatedFunction {  
  
 private final double[] args;  
  
 */\*\**  
 *\* Коэффициенты кубического сплайна: a(i), b(i), c(i) и d(i)*  
 *\*/*  
private final double[][] coefficients;  
  
  
 public CubicSpline(Function function, Section section, double a, double b) {  
 super(section, InterpolationMethod.*CUBIC\_SPLINE*);  
 this.args = section.getSeparation();  
 var fValues = new double[args.length];  
 for (int j = 0; j < args.length; j++) {  
 fValues[j] = function.calculateExpression(args[j]);  
 }  
 coefficients = EquationSolver.*apply*(args, fValues, a, b);  
 }  
  
 @Override  
 public double calculateExpression(double x) {  
  
 var i = locateIndex(x);  
  
 var h\_i = x - args[i];  
  
 return coefficients[i][0]  
 + h\_i \* coefficients[i][1]  
 + 0.5 \* coefficients[i][2] \* Math.*pow*(h\_i, 2)  
 + coefficients[i][3] / 6 \* Math.*pow*(h\_i, 3);  
 }  
  
 */\*\**  
 *\* Вычисление значения n-ой производной сплайна в точке x*  
 *\**  
 *\* @param x Точка*  
 *\* @param derivativeNumber Порядок производной (0+)*  
 *\* @return Значение производной*  
 *\*/*  
public double calculateExpression(double x, int derivativeNumber) {  
  
 if (derivativeNumber < 0) {  
 throw new ArithmeticException("Значение производной не может быть ниже 0");  
 }  
  
 if (derivativeNumber == 0) {  
 return calculateExpression(x);  
 }  
  
 var i = locateIndex(x);  
  
 var h\_i = x - args[i];  
  
 if (derivativeNumber == 1) {  
 return coefficients[i][1] + coefficients[i][2] \* h\_i + 0.5 \* coefficients[i][3] \* Math.*pow*(h\_i, 2);  
 }  
  
 if (derivativeNumber == 2) {  
 return coefficients[i][2] + coefficients[i][3] \* h\_i;  
 }  
  
 if (derivativeNumber == 3) {  
 return coefficients[i][3];  
 }  
  
 return 0;  
 }  
  
 private int locateIndex(double x) {  
  
 var index = args.length / 2;  
  
 int leftBound = 0;  
 int rightBound = args.length;  
 while (  
 index != 0 && index != args.length - 1 &&  
 !(x >= args[index - 1] && x <= args[index])  
 ) {  
 if (x > (args[index])) {  
 leftBound = index;  
 } else {  
 rightBound = index;  
 }  
 index = (leftBound + rightBound) / 2;  
 }  
  
 return index == 0 ? 1 : index;  
 }  
}

**EquationSolver****.****java**

package lab.numeric.methods;  
  
import com.k4r3l1ns.models.TapeMatrix;  
import com.k4r3l1ns.models.Vector;  
import com.k4r3l1ns.service.TapeMatrixOperationUnit;  
  
public class EquationSolver {  
  
 private static final int *SIZE* = 4;  
  
 public static double[][] apply(double[] args, double[] fValues, double a, double b) {  
  
 var h = new double[args.length];  
 for (int i = 1; i < args.length; ++i) {  
 h[i] = args[i] - args[i - 1];  
 }  
  
 var coefficients = new double[args.length][*SIZE*];  
  
 // коэффициенты a: из s(x\_i) = f(x\_i)  
 for (int i = 0; i < args.length; ++i) {  
 coefficients[i][0] = fValues[i];  
 }  
  
 final int tapeWidth = 2;  
 var cCoeffMatrix = new TapeMatrix(args.length, tapeWidth);  
 var matrixValueVector = new Vector((2 \* tapeWidth - 1) \* args.length);  
 for (int i = 3; i < (2 \* tapeWidth - 1) \* args.length - 3; ++i) {  
 int line = i / (2 \* tapeWidth - 1);  
 double value = switch (i % 3) {  
 case 0 -> h[line];  
 case 1 -> (h[line] + h[line + 1]) \* 2;  
 case 2 -> h[line + 1];  
 default -> 0;  
 };  
 matrixValueVector.setValueAt(i, value);  
 }  
 matrixValueVector.setValueAt(1, -1);  
 matrixValueVector.setValueAt(2, 1);  
 matrixValueVector.setValueAt((2 \* tapeWidth - 1) \* args.length - 3, 0.5);  
 matrixValueVector.setValueAt((2 \* tapeWidth - 1) \* args.length - 2, 1);  
  
 cCoeffMatrix.setValues(matrixValueVector);  
  
 var fVector = new Vector(args.length);  
 for (int i = 1; i < args.length - 1; ++i) {  
 var value1 = (fValues[i + 1] - fValues[i]) / h[i + 1];  
 var value2 = (fValues[i] - fValues[i - 1]) / h[i];  
 fVector.setValueAt(i, 6 \* (value1 - value2));  
 }  
 fVector.setValueAt(0, h[1] \* a);  
 fVector.setValueAt(  
 args.length - 1,  
 (b - (fValues[args.length - 1] - fValues[args.length - 2]) / h[args.length - 1])  
 \* 3 / h[args.length - 1]  
 );  
  
 var cCoeffs = TapeMatrixOperationUnit.*solveEquation*(cCoeffMatrix, fVector);  
  
 for (int i = 0; i < args.length; ++i) {  
 coefficients[i][2] = cCoeffs.getValueAt(i);  
 }  
  
 for (int i = 1; i < args.length; ++i) {  
 coefficients[i][3] = (coefficients[i][2] - coefficients[i - 1][2]) / h[i];  
 }  
  
 // { f(i) - f(i-1) - (h[i]^3 / 6 \* coef[i][3]) + (h[i]^2 / 2 \* coef[i][2]) } / h[i]  
 for (int i = 1; i < args.length; ++i) {  
 coefficients[i][1] = (fValues[i] - (fValues[i - 1]) - Math.*pow*(h[i], 3) / 6 \* coefficients[i][3]  
 + Math.*pow*(h[i], 2) \* 0.5 \* coefficients[i][2]) / h[i];  
 }  
  
 return coefficients;  
 }  
}