**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

Лабораторная работа №1

**Решение краевой задачи для обыкновенного дифференциального уравнения второго порядка**

Вариант 2

**Выполнил:**

Кендысь Алексей Максимович

студент 3 курса, 7 группы,

специальность

“прикладная математика”

**Преподаватель:**

Доцент кафедры вычислительной

математики ФПМИ,

А.М. Будник

Минск, 2023

**Содержание:**

Постановка задачи 2

Аппроксимация разностными операторами 2

Краткие теоретические сведения 2

Листинг программы 3

Результаты 6

Выводы 6

Интегро-интерполяционный метод (метод баланса) 7

Краткие теоретические сведения 7

Листинг программы 7

Результаты 9

Выводы 9

Вариационно-разностный метод (метод Ритца) 10

Краткие теоретические сведения 10

Листинг программы 10

Результаты 12

Выводы 13

Постановка задачи

Дана третья краевая задача для ОДУ второго порядка следующего вида:

где

Необходимо:

1. Используя разностные операторы вместо дифференциальных, аппроксимировать поставленную задачу разностной схемой 2-го порядка на минимальном шаблоне. Для повышения порядка аппроксимации граничных условий выделить главный член погрешности и заменить его, используя следующий вид исходного дифференциального уравнения: .
2. Интегро-интерполяционным методом построить консервативную разностную схему. Для вычисления коэффициентов этой схемы использовать формулу трапеций.
3. Аппроксимировать исходную задачу вариационно-разностным методом. Для вычисления коэффициентов полученной схемы использовать формулу средних прямоугольников.
4. Методом разностной прогонки реализовать полученные в п.п. 1-3 разностные схемы при . Провести сравнительный анализ сеточных решений при разных шагах.

Аппроксимация разностными операторами

Краткие теоретические сведения

Рассмотрим разбиение отрезка для (количество разбиений). Величина шага Узлы , *,* внутренние точки разбиения (без ).

Разностная схема, полученная с помощью аппроксимации дифференциальных операторов разностными, записывается в виде (безындексная форма):

где – сеточная функция, , ̶ центральная разностная производная, ̶ правая разностная производная, ̶ левая разностная производная, ,

Здесь

*.*

Запишем индексную форму данной разностной схемы:

Данная схема имеет второй порядок аппроксимации, т.е. .

Покажем вывод формул для и через погрешность для условия на правой границе, .

Тогда, если взять , то получим , т.е. второй порядок аппроксимации.

Для реализации схемы используется метод разностной прогонки. Схема должна быть приведена к виду:

Тогда алгоритм метода:

Приводя нашу схему к нужному виду, получаем выражение для коэффициентов:

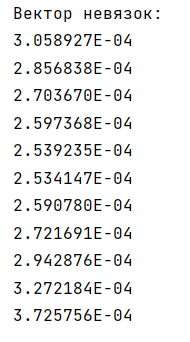
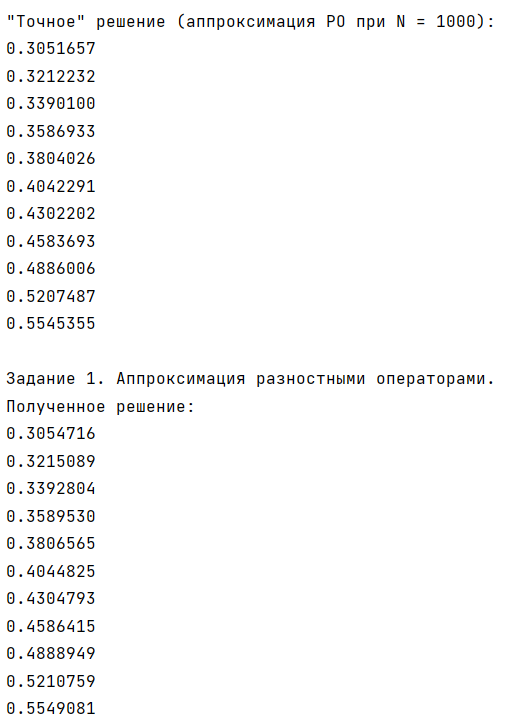
В качестве точного решения используется решение, полученное при использовании данной схемы с .

Листинг программы

import java.util.\*;  
  
class F {  
 public static double getValue(double x) {  
 return Math.*pow*(Math.*sin*(x), 2.);  
 }  
}  
  
class K {  
 public static double getValue(double x) {  
 return Math.*pow*(Math.*cos*(x), 2.) + 1.;  
 }  
  
 public static double getDxValue(double x) {  
 return - Math.*sin*(2. \* x);  
 }  
}  
  
class Q {  
 public static double getValue(double x) {  
 return 1.;  
 }  
}  
  
class Partition {  
 private final int n;  
 private final double h;  
 private final double[] x;  
  
 public Partition(int n, double h) {  
 this.n = n;  
 this.h = h;  
  
 x = new double[n + 1];  
 genX();  
 }  
  
 public double[] getX() {  
 return x;  
 }  
  
 private void genX() {  
 for (int i = 0; i <= n; i++) {  
 x[i] = i \* h;  
 }  
 }  
}  
  
record TridiagMatrixDiffAlgorithm(int n, double[] a, double[] c, double[] b, double[] f, double kap1, double nu1,  
 double kap2, double nu2) {  
  
 public double[] solve() {  
 double[] alpha, beta;  
 alpha = getAlpha();  
 beta = getBeta(alpha);  
  
 double[] y;  
 y = getY(alpha, beta);  
  
 return y;  
 }  
  
 private double[] getAlpha() {  
 double[] res = new double[n];  
 res[0] = kap1;  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i + 1] = b[i] / (c[i] - (a[i] \* res[i]));  
 }  
  
 return res;  
 }  
  
 private double[] getBeta(double[] alpha) {  
 double[] res = new double[n];  
 res[0] = nu1;  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i + 1] = (f[i] + (res[i] \* a[i])) / (c[i] - (a[i] \* alpha[i]));  
 }  
  
 return res;  
 }  
  
 private double[] getY(double[] alpha, double[] beta) {  
 double[] res = new double[n + 1];  
 res[n] = (nu2 + kap2 \* beta[n - 1]) / (1. - (alpha[n - 1] \* kap2));  
  
 for (int i = n - 1; i >= 0; i--) {  
 res[i] = alpha[i] \* res[i + 1] + beta[i];  
 }  
  
 return res;  
 }  
}  
  
record DiffOperatorsMethod (int n, double h, double[] x, double kap0, double g0,  
 double kap1, double g1) implements Method {  
  
 public double[] getA() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = ((K.*getValue*(x[i + 1]) / h) - (K.*getDxValue*(x[i + 1]) / 2.)) / h;  
 }  
  
 return res;  
 }  
  
 public double[] getC() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = ((2. \* K.*getValue*(x[i + 1])) / (h \* h)) + Q.*getValue*(x[i + 1]);  
 }  
  
 return res;  
 }  
  
 public double[] getB() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = ((K.*getValue*(x[i + 1]) / h) + (K.*getDxValue*(x[i + 1]) / 2.)) / h;  
 }  
  
 return res;  
 }  
  
 public double[] getF() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = F.*getValue*(x[i + 1]);  
 }  
  
 return res;  
 }  
  
 public double getNewKap1(double wavyKap0) {  
 return K.*getValue*(x[0]) / ((h \* wavyKap0) + K.*getValue*(x[0]));  
 }  
  
 public double getNu1(double wavyKap0, double wavyG0) {  
 return (h \* wavyG0) / ((h \* wavyKap0) + K.*getValue*(x[0]));  
 }  
  
 public double getWavyKap0() {  
 return (kap0 \* (1 - (h / 2.) \* (K.*getDxValue*(x[0]) / K.*getValue*(x[0])))) + ((h / 2.) \* Q.*getValue*(x[0]));  
 }  
  
 public double getWavyG0() {  
 return (g0 \* (1 - (h / 2.) \* (K.*getDxValue*(x[0]) / K.*getValue*(x[0])))) + ((h / 2.) \* F.*getValue*(x[0]));  
 }  
  
 public double getNewKap2(double wavyKap1) {  
 return K.*getValue*(x[n]) / ((h \* wavyKap1) + K.*getValue*(x[n]));  
 }  
  
 public double getNu2(double wavyKap1, double wavyG1) {  
 return (h \* wavyG1) / ((h \* wavyKap1) + K.*getValue*(x[n]));  
 }  
  
 public double getWavyKap1() {  
 return (kap1 \* (1 + (h / 2.) \* (K.*getDxValue*(x[n]) / K.*getValue*(x[n])))) + ((h / 2.) \* Q.*getValue*(x[n]));  
 }  
  
 public double getWavyG1() {  
 return (g1 \* (1 + (h / 2.) \* (K.*getDxValue*(x[n]) / K.*getValue*(x[n])))) + ((h / 2.) \* F.*getValue*(x[n]));  
 }  
}

class BoundaryValueProblem {  
 private static final double *KAP0* = 1.;  
 private static final double *G0* = 0;  
 private static final double *KAP1* = 1.;  
 private static final double *G1* = 1.;  
 private static final double *H* = 0.1;  
 private static final double *H\_EXACT* = 0.001;  
 private final int n;  
 private final int nExact;  
 private final double[] u;  
 private double[] y1;  
 private double[] res1;   
  
 public BoundaryValueProblem() {  
 n = (int) (1. / *H*);  
 nExact = (int) (1. / *H\_EXACT*);  
  
 Partition exPart = new Partition(nExact, *H\_EXACT*);  
 u = getExact(exPart);  
 }  
  
 public void solve() {  
 Partition p = new Partition(n, *H*);  
 double[] x = p.getX();  
  
 DiffOperatorsMethod dom = new DiffOperatorsMethod(n, *H*, x, *KAP0*, *G0*, *KAP1*, *G1*);  
 y1 = dom.getY(n);  
 res1 = getResidual(y1);   
 }  
  
 public void outRes() {  
 Formatter fmt = new Formatter();  
  
 fmt.format("\n\"Точное\" решение (аппроксимация РО при N = 1000):\n");  
 outY(fmt, u);  
  
 fmt.format("\nЗадание 1. Аппроксимация разностными операторами.\n");  
 fmt.format("Полученное решение:\n");  
 outY(fmt, y1);  
 fmt.format("Вектор невязок:\n");  
 outRes(fmt, res1);  
  
 System.*out*.println(fmt);  
 }  
  
 private double[] getResidual(double[] y) {  
 double[] res = new double[n + 1];  
 for (int i = 0; i <= n; i++) {  
 res[i] = Math.*abs*(u[i] - y[i]);  
 }  
  
 return res;  
 }  
  
 private double[] getExact(Partition p) {  
 double[] res = new double[n + 1];  
 double[] fullRes;  
  
 DiffOperatorsMethod dom = new DiffOperatorsMethod  
 (nExact, *H\_EXACT*, p.getX(), *KAP0*, *G0*, *KAP1*, *G1*);  
 fullRes = dom.getY(nExact);  
  
 for (int i = 0, j = 0; i <= nExact; i += nExact / n, j++) {  
 res[j] = fullRes[i];  
 }  
  
 return res;  
 }  
}  
  
public class Main {  
  
 public static void main(String[] args) {  
 BoundaryValueProblem bvp = new BoundaryValueProblem();  
 bvp.solve();  
 bvp.outRes();  
 }  
}

Результаты



Выводы

Разностная схема, которую мы использовали в данном пункте, имеет второй порядок. Соответственно, погрешность аппроксимации составляет в нашем случае . Из вида вектора невязок (разница между “точным” и полученным решением) можем сказать, что реальная погрешность составляет примерно , т.е. даже на два порядка меньше, чем ожидаемая погрешность – . Результаты вышли лучше ожидаемого, это может быть связано с величиной коэффициента при главном члене погрешности. Также на границах погрешность вышла чуть больше, чем во внутренних точках. Это объясняется тем, что разностные операторы, которые были использованы для аппроксимации граничных условий, обладают первым порядком, т.е. . Второй порядок же был достигнут при помощи подбора констант и из вида погрешности.

Интегро-интерполяционный метод (метод баланса)

Краткие теоретические сведения

Разностная схема для исходной задачи, полученная по методу баланса, записывается в виде (безындексная форма):

где

Здесь

*,*

где

Запишем индексную форму данной разностной схемы:

Данная схема имеет второй порядок аппроксимации, т.е. .

Покажем вывод формул для и с помощью уравнения баланса.

Используем уравнение баланса (отрезок ):

где . Заменим функцию в интеграле полиномом нулевой степени, .

Из правого граничного условия задачи: .

В методе баланса было выведено приближённое равенство

*.* Отсюда получаем: .

Подставляя полученные выражения в уравнение баланса, получаем:

*.*

В итоге имеем:

.

Тогда, если обозначить , то получим записанную ранее аппроксимацию для условия на правой границе.

Для реализации схемы используется метод разностной прогонки.Приводя нашу схему к нужному виду, получаем выражение для коэффициентов:

Для вычисления коэффициентов и используется квадратурная формула трапеций:

которая имеет погрешность .

Листинг программы

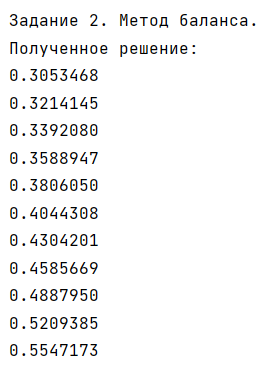
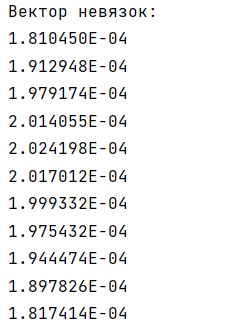
import java.util.\*;  
  
class F {  
 public static double getValue(double x) {  
 return Math.*pow*(Math.*sin*(x), 2.);  
 }  
}  
  
class K {  
 public static double getValue(double x) {  
 return Math.*pow*(Math.*cos*(x), 2.) + 1.;  
 }  
  
 public static double getDxValue(double x) {  
 return - Math.*sin*(2. \* x);  
 }  
}  
  
class Q {  
 public static double getValue(double x) {  
 return 1.;  
 }  
}

class IntegralApprox {  
 public static double TrapezoidalRule (double a, double b, double fA, double fB) {  
 return (b - a) \* ((fA + fB) / 2.);  
 }  
}

abstract class BalanceAndRitzMethod implements Method {  
 protected final int n;  
 protected final double h;  
 protected final double[] x;  
 protected final double kap0;  
 protected final double g0;  
 protected final double kap1;  
 protected final double g1;  
  
 public BalanceAndRitzMethod(int n, double h, double[] x, double kap0, double g0, double kap1, double g1) {  
 this.n = n;  
 this.h = h;  
 this.x = x;  
 this.kap0 = kap0;  
 this.g0 = g0;  
 this.kap1 = kap1;  
 this.g1 = g1;  
 }  
  
 public double[] getA() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = getAI(i + 1) / (h \* h);  
 }  
  
 return res;  
 }  
  
 public double[] getC() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = (getAI(i + 2) + getAI(i + 1)) / (h \* h);  
 res[i] += getDI(i + 1);  
 }  
  
 return res;  
 }  
  
 public double[] getB() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = getAI(i + 2) / (h \* h);  
 }  
  
 return res;  
 }  
  
 public double[] getF() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = getPhiI(i + 1);  
 }  
  
 return res;  
 }  
  
 public double getNewKap1(double wavyKap0) {  
 return getAI(1) / ((h \* wavyKap0) + getAI(1));  
 }  
  
 public double getNu1(double wavyKap0, double wavyG0) {  
 return (h \* wavyG0) / ((h \* wavyKap0) + getAI(1));  
 }  
  
 public double getWavyKap0() {  
 return kap0 + ((h / 2.) \* getD0());  
 }  
  
 public double getWavyG0() {  
 return g0 + ((h / 2.) \* getPhi0());  
 }  
  
 public double getNewKap2(double wavyKap1) {  
 return getAI(n) / ((h \* wavyKap1) + getAI(n));  
 }  
  
 public double getNu2(double wavyKap1, double wavyG1) {  
 return (h \* wavyG1) / ((h \* wavyKap1) + getAI(n));  
 }  
  
 public double getWavyKap1() {  
 return kap1 + ((h / 2.) \* getDN());  
 }  
  
 public double getWavyG1() {  
 return g1 + ((h / 2.) \* getPhiN());  
 }  
  
 protected abstract double getAI(int i);  
 protected abstract double getDI(int i);  
 protected abstract double getPhiI(int i);  
 protected abstract double getD0();  
 protected abstract double getPhi0();  
 protected abstract double getDN();  
 protected abstract double getPhiN();  
}  
  
class BalanceMethod extends BalanceAndRitzMethod {  
  
 public BalanceMethod(int n, double h, double[] x, double kap0, double g0, double kap1, double g1) {  
 super(n, h, x, kap0, g0, kap1, g1);  
 }  
  
 protected double getAI(int i) {  
 double kA = 1 / K.*getValue*(x[i - 1]);  
 double kB = 1 / K.*getValue*(x[i]);  
 return h / IntegralApprox.*TrapezoidalRule*(x[i - 1], x[i], kA, kB);  
 }  
  
 protected double getDI(int i) {  
 double a = x[i] - (h / 2.);  
 double b = x[i] + (h / 2.);  
 double qA = Q.*getValue*(a);  
 double qB = Q.*getValue*(b);  
 return IntegralApprox.*TrapezoidalRule*(a, b, qA, qB) / h;  
 }  
  
 protected double getPhiI(int i) {  
 double a = x[i] - (h / 2.);  
 double b = x[i] + (h / 2.);  
 double fA = F.*getValue*(a);  
 double fB = F.*getValue*(b);  
 return IntegralApprox.*TrapezoidalRule*(a, b, fA, fB) / h;  
 }  
  
 protected double getD0() {  
 double qA = Q.*getValue*(0);  
 double qB = Q.*getValue*(h / 2.);  
 return (2 \* IntegralApprox.*TrapezoidalRule*(0, h / 2., qA, qB)) / h;  
 }  
  
 protected double getPhi0() {  
 double fA = F.*getValue*(0);  
 double fB = F.*getValue*(h / 2.);  
 return (2 \* IntegralApprox.*TrapezoidalRule*(0, h / 2., fA, fB)) / h;  
 }  
  
 protected double getDN() {  
 double a = 1 - (h / 2.);  
 double b = 1.;  
 double qA = Q.*getValue*(a);  
 double qB = Q.*getValue*(b);  
 return (2 \* IntegralApprox.*TrapezoidalRule*(a, b, qA, qB)) / h;  
 }  
  
 protected double getPhiN() {  
 double a = 1 - (h / 2.);  
 double b = 1.;  
 double fA = F.*getValue*(a);  
 double fB = F.*getValue*(b);  
 return (2 \* IntegralApprox.*TrapezoidalRule*(a, b, fA, fB)) / h;  
 }  
}

class BoundaryValueProblem {  
 private static final double *KAP0* = 1.;  
 private static final double *G0* = 0;  
 private static final double *KAP1* = 1.;  
 private static final double *G1* = 1.;  
 private static final double *H* = 0.1;   
 private final int n;   
 private double[] y2;  
 private double[] res2;   
  
 public BoundaryValueProblem() {  
 n = (int) (1. / *H*);   
 }  
  
 public void solve() {  
 Partition p = new Partition(n, *H*);  
 double[] x = p.getX();  
  
 BalanceMethod bm = new BalanceMethod(n, *H*, x, *KAP0*, *G0*, *KAP1*, *G1*);  
 y2 = bm.getY(n);  
 res2 = getResidual(y2);   
 }  
  
 public void outRes() {  
 Formatter fmt = new Formatter();  
  
 fmt.format("\nЗадание 2. Метод баланса.\n");  
 fmt.format("Полученное решение:\n");  
 outY(fmt, y2);  
 fmt.format("Вектор невязок:\n");  
 outRes(fmt, res2);   
  
 System.*out*.println(fmt);  
 }  
  
 private double[] getResidual(double[] y) {  
 double[] res = new double[n + 1];  
 for (int i = 0; i <= n; i++) {  
 res[i] = Math.*abs*(u[i] - y[i]);  
 }  
  
 return res;  
 }  
}  
  
public class Main {  
  
 public static void main(String[] args) {  
 BoundaryValueProblem bvp = new BoundaryValueProblem();  
 bvp.solve();  
 bvp.outRes();  
 }  
}

Результаты

Выводы

Как и для прошлого метода, разностная схема, которую мы использовали в данном пункте, имеет второй порядок. Соответственно, погрешность аппроксимации составляет в нашем случае . Из вида вектора невязок сказать, что реальная погрешность составляет примерно , т.е. снова на два порядка меньше, чем ожидаемая погрешность – . Результаты вышли лучше ожидаемого, это может быть связано с величиной коэффициента при главном члене погрешности.

Несмотря на то, что при применении метода интегралы были вычислены приближённо, погрешность получилась такая же, как и для прошлого метода. Используемая формула трапеций имеет АСТ = 1, погрешность имеет вид . В коэффициентах также есть множитель , поэтому в итоге получается погрешность . Причиной, почему эта погрешность никак не влияет на результаты, является то, что подынтегральные функции, для которых вычислялись интегралы, имеют достаточно гладкие производные, которые принимают небольшие значения на отрезке [0;1]. А т.к. производные присутствуют в остатке квадратурной формулы, то коэффициент при главном члене погрешности довольно мал (не увеличивает порядок в случае ).

Вариационно-разностный метод (метод Ритца)

Краткие теоретические сведения

Разностная схема по методу Ритца имеет тот же вид, что и схема для метода баланса. Отличие только в вычислении коэффициентов . Запишем их:

Данная схема имеет второй порядок аппроксимации, т.е. .

Для вычисления коэффициентов и используется квадратурная формула средних прямоугольников:

которая имеет погрешность .

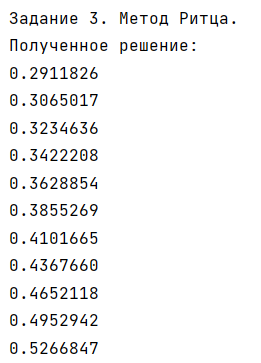
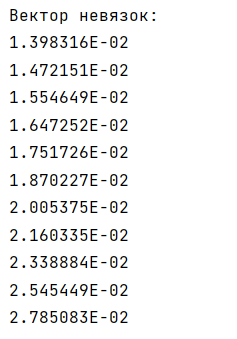
Листинг программы

import java.util.\*;  
  
class F {  
 public static double getValue(double x) {  
 return Math.*pow*(Math.*sin*(x), 2.);  
 }  
}  
  
class K {  
 public static double getValue(double x) {  
 return Math.*pow*(Math.*cos*(x), 2.) + 1.;  
 }  
  
 public static double getDxValue(double x) {  
 return - Math.*sin*(2. \* x);  
 }  
}  
  
class Q {  
 public static double getValue(double x) {  
 return 1.;  
 }  
}

class IntegralApprox {  
 public static double TrapezoidalRule (double a, double b, double fA, double fB) {  
 return (b - a) \* ((fA + fB) / 2.);  
 }  
}

abstract class BalanceAndRitzMethod implements Method {  
 protected final int n;  
 protected final double h;  
 protected final double[] x;  
 protected final double kap0;  
 protected final double g0;  
 protected final double kap1;  
 protected final double g1;  
  
 public BalanceAndRitzMethod(int n, double h, double[] x, double kap0, double g0, double kap1, double g1) {  
 this.n = n;  
 this.h = h;  
 this.x = x;  
 this.kap0 = kap0;  
 this.g0 = g0;  
 this.kap1 = kap1;  
 this.g1 = g1;  
 }  
  
 public double[] getA() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = getAI(i + 1) / (h \* h);  
 }  
  
 return res;  
 }  
  
 public double[] getC() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = (getAI(i + 2) + getAI(i + 1)) / (h \* h);  
 res[i] += getDI(i + 1);  
 }  
  
 return res;  
 }  
  
 public double[] getB() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = getAI(i + 2) / (h \* h);  
 }  
  
 return res;  
 }  
  
 public double[] getF() {  
 double[] res = new double[n - 1];  
  
 for (int i = 0; i < n - 1; i++) {  
 res[i] = getPhiI(i + 1);  
 }  
  
 return res;  
 }  
  
 public double getNewKap1(double wavyKap0) {  
 return getAI(1) / ((h \* wavyKap0) + getAI(1));  
 }  
  
 public double getNu1(double wavyKap0, double wavyG0) {  
 return (h \* wavyG0) / ((h \* wavyKap0) + getAI(1));  
 }  
  
 public double getWavyKap0() {  
 return kap0 + ((h / 2.) \* getD0());  
 }  
  
 public double getWavyG0() {  
 return g0 + ((h / 2.) \* getPhi0());  
 }  
  
 public double getNewKap2(double wavyKap1) {  
 return getAI(n) / ((h \* wavyKap1) + getAI(n));  
 }  
  
 public double getNu2(double wavyKap1, double wavyG1) {  
 return (h \* wavyG1) / ((h \* wavyKap1) + getAI(n));  
 }  
  
 public double getWavyKap1() {  
 return kap1 + ((h / 2.) \* getDN());  
 }  
  
 public double getWavyG1() {  
 return g1 + ((h / 2.) \* getPhiN());  
 }  
  
 protected abstract double getAI(int i);  
 protected abstract double getDI(int i);  
 protected abstract double getPhiI(int i);  
 protected abstract double getD0();  
 protected abstract double getPhi0();  
 protected abstract double getDN();  
 protected abstract double getPhiN();  
}  
  
class RitzMethod extends BalanceAndRitzMethod {  
  
 public RitzMethod(int n, double h, double[] x, double kap0, double g0, double kap1, double g1) {  
 super(n, h, x, kap0, g0, kap1, g1);  
 }  
  
 protected double getAI(int i) {  
 double midpoint;  
 double fValue1, fValue2;  
 double iValue1, iValue2;  
  
 midpoint = (x[i - 1] + x[i]) / 2.;  
  
 fValue1 = K.*getValue*(midpoint);  
 iValue1 = IntegralApprox.*MidpointRule*(x[i - 1], x[i], fValue1);  
  
 fValue2 = Q.*getValue*(midpoint) \* (midpoint - x[i - 1]) \* (x[i] - midpoint);  
 iValue2 = IntegralApprox.*MidpointRule*(x[i - 1], x[i], fValue2);  
  
 return (iValue1 - iValue2) / h;  
 }  
  
 protected double getDI(int i) {  
 double midpoint1, midpoint2;  
 double fValue1, fValue2;  
 double iValue1, iValue2;  
  
 midpoint1 = (x[i - 1] + x[i]) / 2.;  
 midpoint2 = (x[i] + x[i + 1]) / 2.;  
  
 fValue1 = Q.*getValue*(midpoint1) \* (midpoint1 - x[i - 1]);  
 iValue1 = IntegralApprox.*MidpointRule*(x[i - 1], x[i], fValue1);  
  
 fValue2 = Q.*getValue*(midpoint2) \* (x[i + 1] - midpoint2);  
 iValue2 = IntegralApprox.*MidpointRule*(x[i], x[i + 1], fValue2);  
  
 return (iValue1 + iValue2) / (h \* h);  
 }  
  
 protected double getPhiI(int i) {  
 double midpoint1, midpoint2;  
 double fValue1, fValue2;  
 double iValue1, iValue2;  
  
 midpoint1 = (x[i - 1] + x[i]) / 2.;  
 midpoint2 = (x[i] + x[i + 1]) / 2.;  
  
 fValue1 = F.*getValue*(midpoint1) \* (midpoint1 - x[i - 1]);  
 iValue1 = IntegralApprox.*MidpointRule*(x[i - 1], x[i], fValue1);  
  
 fValue2 = F.*getValue*(midpoint2) \* (x[i + 1] - midpoint2);  
 iValue2 = IntegralApprox.*MidpointRule*(x[i], x[i + 1], fValue2);  
  
 return (iValue1 + iValue2) / (h \* h);  
 }  
  
 protected double getD0() {  
 double midpoint;  
 double fValue, iValue;  
  
 midpoint = h / 2.;  
 fValue = Q.*getValue*(midpoint) \* (h - midpoint);  
 iValue = IntegralApprox.*MidpointRule*(0, h, fValue);  
  
 return (2. \* iValue) / (h \* h);  
 }  
  
 protected double getPhi0() {  
 double midpoint;  
 double fValue, iValue;  
  
 midpoint = h / 2.;  
 fValue = F.*getValue*(midpoint) \* (h - midpoint);  
 iValue = IntegralApprox.*MidpointRule*(0, h, fValue);  
  
 return (2. \* iValue) / (h \* h);  
 }  
  
 protected double getDN() {  
 double midpoint;  
 double fValue, iValue;  
  
 midpoint = (1. - h + 1.) / 2.;  
 fValue = Q.*getValue*(midpoint) \* (midpoint - 1. + h);  
 iValue = IntegralApprox.*MidpointRule*(1. - h, h, fValue);  
  
 return (2. \* iValue) / (h \* h);  
 }  
  
 protected double getPhiN() {  
 double midpoint;  
 double fValue, iValue;  
  
 midpoint = (1. - h + 1.) / 2.;  
 fValue = F.*getValue*(midpoint) \* (midpoint - 1. + h);  
 iValue = IntegralApprox.*MidpointRule*(1. - h, h, fValue);  
  
 return (2. \* iValue) / (h \* h);  
 }  
}  
  
class BoundaryValueProblem {  
 private static final double *KAP0* = 1.;  
 private static final double *G0* = 0;  
 private static final double *KAP1* = 1.;  
 private static final double *G1* = 1.;  
 private static final double *H* = 0.1;   
 private final int n;   
 private double[] y3;  
 private double[] res3;  
  
 public BoundaryValueProblem() {  
 n = (int) (1. / *H*);   
 }  
  
 public void solve() {  
 Partition p = new Partition(n, *H*);  
 double[] x = p.getX();  
  
 RitzMethod rm = new RitzMethod(n, *H*, x, *KAP0*, *G0*, *KAP1*, *G1*);  
 y3 = rm.getY(n);  
 res3 = getResidual(y3);  
 }  
  
 public void outRes() {  
 Formatter fmt = new Formatter();  
  
 fmt.format("\nЗадание 3. Метод Ритца.\n");  
 fmt.format("Полученное решение:\n");  
 outY(fmt, y3);  
 fmt.format("Вектор невязок:\n");  
 outRes(fmt, res3);  
  
 System.*out*.println(fmt);  
 }  
  
 private double[] getResidual(double[] y) {  
 double[] res = new double[n + 1];  
 for (int i = 0; i <= n; i++) {  
 res[i] = Math.*abs*(u[i] - y[i]);  
 }  
  
 return res;  
 }  
}  
  
public class Main {  
  
 public static void main(String[] args) {  
 BoundaryValueProblem bvp = new BoundaryValueProblem();  
 bvp.solve();  
 bvp.outRes();  
 }  
}

Результаты

Выводы

Как и для прошлого метода, разностная схема, которую мы использовали в данном пункте, имеет второй порядок. Соответственно, погрешность аппроксимации составляет в нашем случае . Из вида вектора невязок сказать, что реальная погрешность составляет примерно , т.е. полностью совпадает с ожидаемой погрешностью.

При применении метода интегралы были вычислены приближённо. Погрешность получилась хуже, чем для двух прошлых методов. Используемая формула средних прямоугольников имеет АСТ = 1, как и в прошлом методе. Причиной, почему погрешность вышла хуже, может быть то, что подынтегральные функции, для которых вычислялись интегралы, имеют производные с большими значениями на отрезке [0;1], чем для прошлого метода. А т.к. производные присутствуют в остатке квадратурной формулы, то коэффициент при главном члене погрешности, соответственно, тоже больше, чем для метода баланса. В итоге из-за приближённого вычисления интегралов получается большая погрешность, чем для прошлых методов, но она все равно соответствует ожидаемой.

В конечном итоге получили, что лучше всего сработал метод аппроксимации разностными операторами и метод баланса.