БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Лабораторная работа №1

**Решение краевой задачи для обыкновенного дифференциального уравнения 2 порядка**

Вариант 3

**Выполнила:**

Зуйкевич Лидия

3 курс 7 группа

**Преподаватель:**

Будник А.М.

Минск, 2023

Оглавление

Постановка задачи 3

[**1. Замена дифференциальных операторов разностными** 4](#_Toc118753263)

[Алгоритм решения 4](#_Toc118753265)

[Листинг программы 5](#_Toc118753266)

[Вывод программы 8](#_Toc104902787)

[Выводы 8](#_Toc104902787)

[**2. Интегро-интерполяционный метод** 8](#_Toc118753267)

[Алгоритм решения 8](#_Toc118753269)

[Листинг программы 10](#_Toc118753270)

[Вывод программы 13](#_Toc104902787)

[Выводы 13](#_Toc104902787)

[**3. Метод Ритца** 14](#_Toc118753273)

[Постановка задачи 14](#_Toc118753274)

[Алгоритм решения 14](#_Toc118753275)

[Листинг программы 15](#_Toc118753276)

[Вывод программы 18](#_Toc104902787)

[Выводы 18](#_Toc104902787)

# 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 наилучшую консервативную разностную схему, когда интегралы в коэффициентах этой схемы вычисляются точно.

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.

## 1. Замена дифференциальных операторов разностными

# Алгоритм решения

1) Зададим сетку , и построим на ней схему следующего вида, заменив дифференциальные операторы разностными:

Основное уравнение аппроксимируется с порядком , для того, чтобы аппроксимация условий также имела порядок , выберем значения :

Для левого граничного условия:

Рассмотрим погрешность аппроксимации правого граничного условия:

=

Для достижения заданного порядка аппроксимации выберем:

,

Запишем полученную схему в индексной форме:

Чтобы использовать метод разностной прогонки, приведем схему к следующему виду:

Формулы для коэффициентов прогонки:

Формулы прогонки для нахождения решения:

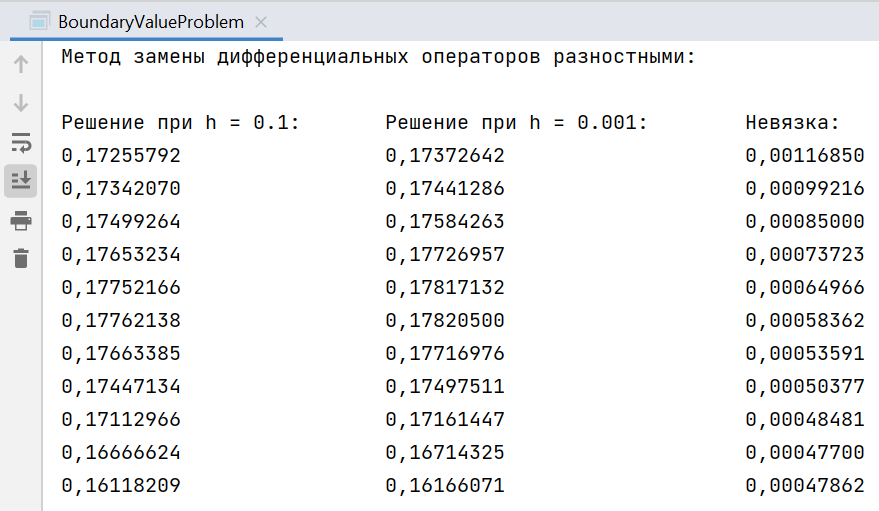
# Листинг программы

public class BoundaryValueProblem {  
 private double h;  
 private int n;  
 private double kappa0;  
 private double kappa1;  
 private double g0;  
 private double g1;  
 private double[] y1;  
 private double[] y2;  
 private double[] y3;  
  
 public BoundaryValueProblem(double h) {  
 this.h = h;  
 this.n = (int) (1 / h);  
 this.kappa0 = 0;  
 this.kappa1 = 1;  
 this.g0 = 0;  
 this.g1 = 0;  
 this.y1 = new double[n + 1];  
 this.y2 = new double[n + 1];  
 this.y3 = new double[n + 1];  
 }  
  
 public double f(double x){  
 return Math.*sin*(x);  
 }  
  
 public double k(double x){  
 return Math.*exp*(x);  
 }  
  
 public double q(double x){  
 return Math.*exp*(x);  
 }  
  
 public double k\_(double x){  
 return Math.*exp*(x);  
 }  
  
 public double[] calculate\_alpha(double[] a, double[] b, double[] c, double k1, int N){  
 double[] alpha = new double[N];  
  
 alpha[0] = k1;  
 for (int i = 0; i < N - 1; i++) {  
 alpha[i + 1] = b[i] / (c[i] - a[i] \* alpha[i]);  
 }  
  
 return alpha;  
 }  
  
 public double[] calculate\_beta(double[] a, double[] c, double[] F, double[] alpha, double v1, int N){  
 double[] beta = new double[N];  
  
 beta[0] = v1;  
 for (int i = 0; i < N - 1; i++) {  
 beta[i + 1] = (F[i] + beta[i] \* a[i]) / (c[i] - a[i] \* alpha[i]);  
 }  
  
 return beta;  
 }  
  
 public double[] tridiagonal\_matrix\_algorithm(double[] a, double[] b, double[] c, double[] F, double k1, double k2, double v1, double v2, int N){  
 double[] y = new double[N + 1];  
 double[] alpha = calculate\_alpha(a, b, c, k1, N);  
 double[] beta = calculate\_beta(a, c, F, alpha, v1, N);  
  
 y[N] = (v2 + k2 \* beta[N - 1]) / (1 - alpha[N - 1] \* k2);  
  
 for (int i = N - 1; i >= 0; i--){  
 y[i] = alpha[i] \* y[i + 1] + beta[i];  
 }  
  
 return y;  
 }  
  
 public double[] differenceOperatorsMethod(double step){  
 int N = (int) (1 / step);  
  
 double[] a = new double[N - 1];  
 double[] b = new double[N - 1];  
 double[] c = new double[N - 1];  
 double[] F = new double[N - 1];  
  
 double val = step;  
 double h2 = Math.*pow*(step, 2);  
  
 for (int i = 0; i < N - 1; i++) {  
 a[i] = k(val) / h2 - k\_(val) / (2 \* step);  
 b[i] = k\_(val) / (2 \* step) + k(val) / h2;  
 c[i] = 2 \* k(val) / h2 + q(val);  
 F[i] = f(val);  
 val += step;  
 }  
  
 double k0\_ = kappa0 \* (1 - (step \* k\_(0)) / (k(0) \* 2)) + (step \* q(0)) / 2;  
 double g0\_ = g0 \* (1 - (k\_(0) \* step) / (k(0) \* 2)) + (step \* f(0)) / 2;  
 double k1\_ = kappa1 \* (1 + (k\_(1) \* step) / (2 \* k(1))) + (step \* q(1)) / 2;  
 double g1\_ = g1 \* (1 + (step \* k\_(1)) / (2 \* k(1))) + (step \* f(1)) / 2;  
 double k1 = k(0) / (step \* k0\_ + k(0));  
 double v1 = (step \* g0\_) / (step \* k0\_ + k(0));  
 double k2 = k(1) / (step \* k1\_ + k(1));  
 double v2 = (step \* g1\_) / (step \* k1\_ + k(1));  
 double[] y = tridiagonal\_matrix\_algorithm(a, b, c, F, k1, k2, v1, v2, N);  
  
 return y;  
 }

public void printArrays(double[] y, double[] u, int h\_diff){  
 double res;  
 System.*out*.print("Решение при h = 0.1: ");  
 System.*out*.print("Решение при h = 0.001: ");  
 System.*out*.print("Невязка: \n");  
 for(int i = 0; i < y.length; i++){  
 res = Math.*abs*(u[i \* h\_diff] - y[i]);  
 System.*out*.print(String.*format*("%5.8f", y[i]));  
 System.*out*.print(String.*format*("%27.8f", u[i \* h\_diff]));  
 System.*out*.print(String.*format*("%30.8f", res));  
 System.*out*.println();  
 }  
}  
  
public void solve(){  
 System.*out*.println("Метод замены дифференциальных операторов разностными: \n");  
 y1 = differenceOperatorsMethod(h);  
 double[] u1 = differenceOperatorsMethod(h / 100);  
 int h\_diff = 100;  
  
 printArrays(y1, u1, h\_diff);}

public static void main(String[] args){  
 BoundaryValueProblem obj = new BoundaryValueProblem(0.1);  
 obj.solve();  
 }  
}

Вывод программы



Выводы

Схема имеет порядок аппроксимации , т.е. погрешность решения должна быть порядка . При сравнении решений с шагами и (решение с шагом взято в качестве точного), видим, что невязка имеет порядок , что даже лучше, чем ожидаемая погрешность, однако, мы сравниваем не с точным решением, а с решением с меньшим шагом.

## 2. Интегро-интерполяционный метод

# Алгоритм решения

Разностная схема для нашей задачи, построенная с помощью интегро-интерполяционного метода (метода баланса) имеет следующий вид:

Коэффициенты схемы:

Для левого граничного условия:

Где

Значения выведем с помощью уравнения баланса на отрезке [1 - ; 1]:

Заменим функцию полиномом нулевой степени: .

Из правого граничного условия: , а также используем равенство т.е. , тогда подставив в уравнение баланса:

Тогда для правого граничного условия:

Где .

Интегралы в коэффициентах по условию считаются точно. Полученная схема имеет второй порядок аппроксимации, т.е. уравнение и условие аппроксимируются с порядком .

Запишем полученную схему в индексной форме:

Чтобы использовать метод разностной прогонки, приведем схему к следующему виду:

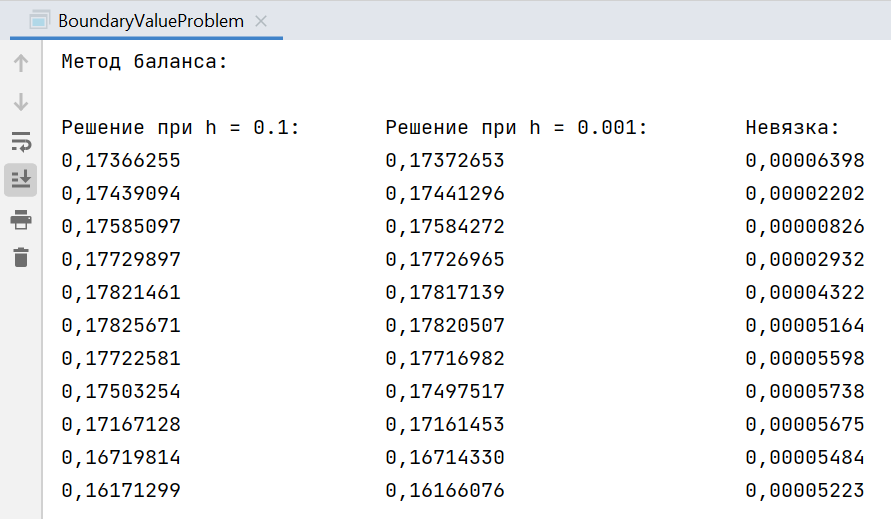
# Листинг программы

public class BoundaryValueProblem {  
 private double h;  
 private int n;  
 private double kappa0;  
 private double kappa1;  
 private double g0;  
 private double g1;  
 private double[] y1;  
 private double[] y2;  
 private double[] y3;  
  
 public BoundaryValueProblem(double h) {  
 this.h = h;  
 this.n = (int) (1 / h);  
 this.kappa0 = 0;  
 this.kappa1 = 1;  
 this.g0 = 0;  
 this.g1 = 0;  
 this.y1 = new double[n + 1];  
 this.y2 = new double[n + 1];  
 this.y3 = new double[n + 1];  
 }  
  
 public double f(double x){  
 return Math.*sin*(x);  
 }  
  
 public double k(double x){  
 return Math.*exp*(x);  
 }  
  
 public double q(double x){  
 return Math.*exp*(x);  
 }  
  
 public double k\_(double x){  
 return Math.*exp*(x);  
 }  
  
 public double[] calculate\_alpha(double[] a, double[] b, double[] c, double k1, int N){  
 double[] alpha = new double[N];  
  
 alpha[0] = k1;  
 for (int i = 0; i < N - 1; i++) {  
 alpha[i + 1] = b[i] / (c[i] - a[i] \* alpha[i]);  
 }  
  
 return alpha;  
 }  
  
 public double[] calculate\_beta(double[] a, double[] c, double[] F, double[] alpha, double v1, int N){  
 double[] beta = new double[N];  
  
 beta[0] = v1;  
 for (int i = 0; i < N - 1; i++) {  
 beta[i + 1] = (F[i] + beta[i] \* a[i]) / (c[i] - a[i] \* alpha[i]);  
 }  
  
 return beta;  
 }  
  
 public double[] tridiagonal\_matrix\_algorithm(double[] a, double[] b, double[] c, double[] F, double k1, double k2, double v1, double v2, int N){  
 double[] y = new double[N + 1];  
 double[] alpha = calculate\_alpha(a, b, c, k1, N);  
 double[] beta = calculate\_beta(a, c, F, alpha, v1, N);  
  
 y[N] = (v2 + k2 \* beta[N - 1]) / (1 - alpha[N - 1] \* k2);  
  
 for (int i = N - 1; i >= 0; i--){  
 y[i] = alpha[i] \* y[i + 1] + beta[i];  
 }  
  
 return y;  
 }

public double[] balanceMethod(double step){  
 int N = (int) (1 / step);  
  
 double[] a = new double[N];  
 double[] d = new double[N + 1];  
 double[] phi = new double[N + 1];  
 double[] A = new double[N - 1];  
 double[] b = new double[N - 1];  
 double[] c = new double[N - 1];  
 double[] F = new double[N - 1];  
  
 double h2 = Math.*pow*(step, 2);  
  
 for (int i = 0; i < N - 1; i++) {  
 a[i] = - step / (Math.*exp*(-(i + 1) \* step) - Math.*exp*(-i \* step));  
 d[i + 1] = (Math.*exp*((i + 1.5) \* step) - Math.*exp*((i + 0.5) \* step)) / step;  
 phi[i + 1] = - (Math.*cos*((i + 1.5) \* step) - Math.*cos*((i + 0.5) \* step)) / step;  
 }  
  
 d[0] = 2 \* (Math.*exp*(step / 2) - 1) / step;  
 d[N] = 2 \* (Math.*exp*(1) - Math.*exp*(1 - step / 2)) / step;  
 phi[0] = - 2 \* (Math.*cos*(step / 2) - 1) / step;  
 phi[N] = - 2 \* (Math.*cos*(1) - Math.*cos*(1 - step / 2)) / step;  
 a[N - 1] = - step / (Math.*exp*(- N \* step) - Math.*exp*(-(N - 1) \* step));  
  
 for (int i = 0; i < N - 1; i++) {  
 A[i] = a[i] / h2;  
 b[i] = a[i + 1] / h2;  
 c[i] = a[i + 1] / h2 + a[i] / h2 + d[i + 1];  
 F[i] = phi[i + 1];  
 }  
  
 double k0\_ = kappa0 + step \* d[0] / 2;  
 double g0\_ = g0 + step \* phi[0] / 2;  
 double k1\_ = kappa1 + step \* d[N] / 2;  
 double g1\_ = g1 + step \* phi[N] / 2;  
 double k1 = a[0] / (step \* k0\_ + a[0]);  
 double v1 = step \* g0\_ / (step \* k0\_ + a[0]);  
 double k2 = a[N - 1] / (step \* k1\_ + a[N - 1]);  
 double v2 = step \* g1\_ / (step \* k1\_ + a[N - 1]);  
 double[] y = tridiagonal\_matrix\_algorithm(A, b, c, F, k1, k2, v1, v2, N);  
  
 return y;  
}

public void printArrays(double[] y, double[] u, int h\_diff){  
 double res;  
 System.*out*.print("Решение при h = 0.1: ");  
 System.*out*.print("Решение при h = 0.001: ");  
 System.*out*.print("Невязка: \n");  
 for(int i = 0; i < y.length; i++){  
 res = Math.*abs*(u[i \* h\_diff] - y[i]);  
 System.*out*.print(String.*format*("%5.8f", y[i]));  
 System.*out*.print(String.*format*("%27.8f", u[i \* h\_diff]));  
 System.*out*.print(String.*format*("%30.8f", res));  
 System.*out*.println();  
 }  
 }  
  
 public void solve(){  
 System.*out*.println("\nМетод баланса: \n");  
 y2 = balanceMethod(h);  
 double[] u2 = balanceMethod(h / 100);  
  
 printArrays(y2, u2, h\_diff);  
 }  
  
 public static void main(String[] args){  
 BoundaryValueProblem obj = new BoundaryValueProblem(0.1);  
 obj.solve();  
 }  
}

Вывод программы



Выводы

Данная схема также имеет порядок ), т.е. погрешность решения должна быть порядка . При сравнении решений с шагами и (решение с шагом взято в качестве точного), видим, что невязка имеет порядок , что лучше ожидаемой погрешности и в среднем на порядок лучше, чем погрешность решения, полученная с помощью первой схемы. Полученный результат можно объяснить тем, что в построенной схеме интегралы в коэффициентах считаются точно, т.е. она является наилучшей консервативной однородной разностной схемой.

## 3. Метод Ритца

# Алгоритм решения

Разностная схема для исходной задачи, построенная с помощью вариационно-разностного метода Ритца имеет тот же вид, что и схема, построенная методом баланса:

Однако коэффициенты схемы вычисляются иначе:

По заданию интегралы в коэффициентах вычисляются с помощью квадратурной формулы средних прямоугольников, которая имеет вид:

Ее погрешность аппроксимации: .

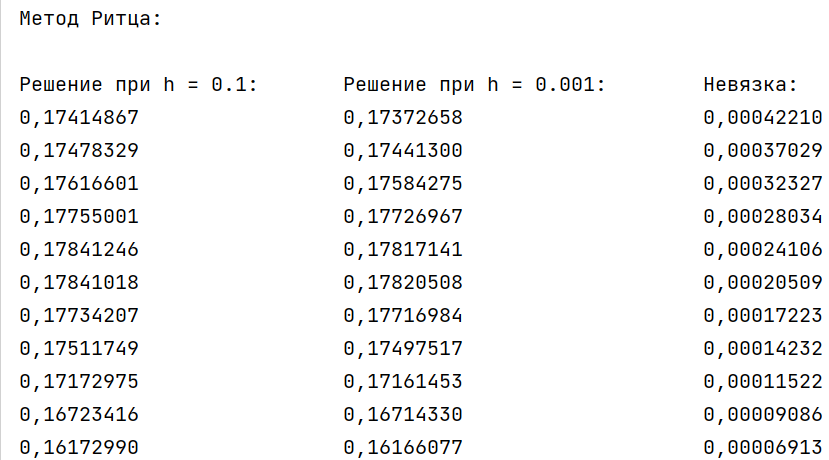
Полученная схема имеет второй порядок аппроксимации, т.е. уравнение и условие аппроксимируются с порядком .

# Листинг программы

public class BoundaryValueProblem {  
 private double h;  
 private int n;  
 private double kappa0;  
 private double kappa1;  
 private double g0;  
 private double g1;  
 private double[] y1;  
 private double[] y2;  
 private double[] y3;  
  
 public BoundaryValueProblem(double h) {  
 this.h = h;  
 this.n = (int) (1 / h);  
 this.kappa0 = 0;  
 this.kappa1 = 1;  
 this.g0 = 0;  
 this.g1 = 0;  
 this.y1 = new double[n + 1];  
 this.y2 = new double[n + 1];  
 this.y3 = new double[n + 1];  
 }  
  
 public double f(double x){  
 return Math.*sin*(x);  
 }  
  
 public double k(double x){  
 return Math.*exp*(x);  
 }  
  
 public double q(double x){  
 return Math.*exp*(x);  
 }  
  
 public double k\_(double x){  
 return Math.*exp*(x);  
 }  
  
 public double[] calculate\_alpha(double[] a, double[] b, double[] c, double k1, int N){  
 double[] alpha = new double[N];  
  
 alpha[0] = k1;  
 for (int i = 0; i < N - 1; i++) {  
 alpha[i + 1] = b[i] / (c[i] - a[i] \* alpha[i]);  
 }  
  
 return alpha;  
 }  
  
 public double[] calculate\_beta(double[] a, double[] c, double[] F, double[] alpha, double v1, int N){  
 double[] beta = new double[N];  
  
 beta[0] = v1;  
 for (int i = 0; i < N - 1; i++) {  
 beta[i + 1] = (F[i] + beta[i] \* a[i]) / (c[i] - a[i] \* alpha[i]);  
 }  
  
 return beta;  
 }  
  
 public double[] tridiagonal\_matrix\_algorithm(double[] a, double[] b, double[] c, double[] F, double k1, double k2, double v1, double v2, int N){  
 double[] y = new double[N + 1];  
 double[] alpha = calculate\_alpha(a, b, c, k1, N);  
 double[] beta = calculate\_beta(a, c, F, alpha, v1, N);  
  
 y[N] = (v2 + k2 \* beta[N - 1]) / (1 - alpha[N - 1] \* k2);  
  
 for (int i = N - 1; i >= 0; i--){  
 y[i] = alpha[i] \* y[i + 1] + beta[i];  
 }  
  
 return y;  
 }

public double sp\_a(double a, double b, double step){  
 double mid = (b + a) / 2;  
 return (b - a) \* (k(mid) - q(mid) \* (mid - a) \* (b - mid)) / step;  
 }  
  
 public double sp\_di(double a, double b, double c, double step){  
 double mid1 = (b + a) / 2;  
 double mid2 = (b + c) / 2;  
 return ((b - a) \* (q(mid1) \* (mid1 - a)) + (c - b) \* (q(mid2) \* (c - mid2))) / Math.*pow*(step, 2);  
 }  
  
 public double sp\_phi\_i(double a, double b,double c, double step){  
 double mid1 = (b + a) / 2;  
 double mid2 = (b + c) / 2;  
 return ((b - a) \* (f(mid1) \* (mid1 - a)) + (c - b) \* (f(mid2) \* (c - mid2))) / Math.*pow*(step, 2);  
 }  
  
 public double[] ritzMethod(double step){  
 int N = (int) (1 / step);  
  
 double[] a = new double[N];  
 double[] d = new double[N + 1];  
 double[] phi = new double[N + 1];  
 double[] A = new double[N - 1];  
 double[] b = new double[N - 1];  
 double[] c = new double[N - 1];  
 double[] F = new double[N - 1];  
  
 double h2 = Math.*pow*(step, 2);  
  
 for (int i = 1; i < N; i++) {  
 a[i - 1] = sp\_a((i - 1) \* step, i \* step, step);  
 d[i] = sp\_di((i - 1) \* step, i \* step, (i + 1) \* step, step);  
 phi[i] = sp\_phi\_i((i - 1) \* step, i \* step, (i + 1) \* step, step);  
 }  
  
 d[0] = q(step / 2);  
 d[N] = q(1 - step / 2);  
 phi[0] = f(step / 2);  
 phi[N] = f(1 - step / 2);  
 a[N - 1] = sp\_a((N - 1) \* step, N \* step, step);  
  
 for (int i = 0; i < N - 1; i++) {  
 A[i] = a[i] / h2;  
 b[i] = a[i + 1] / h2;  
 c[i] = a[i + 1] / h2 + a[i] / h2 + d[i + 1];  
 F[i] = phi[i + 1];  
 }  
  
 double k0\_ = kappa0 + step \* d[0] / 2;  
 double g0\_ = g0 + step \* phi[0] / 2;  
 double k1\_ = kappa1 + step \* d[N] / 2;  
 double g1\_ = g1 + step \* phi[N] / 2;  
 double k1 = a[0] / (step \* k0\_ + a[0]);  
 double v1 = step \* g0\_ / (step \* k0\_ + a[0]);  
 double k2 = a[N - 1] / (step \* k1\_ + a[N - 1]);  
 double v2 = step \* g1\_ / (step \* k1\_ + a[N - 1]);  
 double[] y = tridiagonal\_matrix\_algorithm(A, b, c, F, k1, k2, v1, v2, N);  
  
 return y;  
 }  
  
 public void printArrays(double[] y, double[] u, int h\_diff){  
 double res;  
 System.*out*.print("Решение при h = 0.1: ");  
 System.*out*.print("Решение при h = 0.001: ");  
 System.*out*.print("Невязка: \n");  
 for(int i = 0; i < y.length; i++){  
 res = Math.*abs*(u[i \* h\_diff] - y[i]);  
 System.*out*.print(String.*format*("%5.8f", y[i]));  
 System.*out*.print(String.*format*("%27.8f", u[i \* h\_diff]));  
 System.*out*.print(String.*format*("%30.8f", res));  
 System.*out*.println();  
 }  
 }  
  
 public void solve(){  
 System.*out*.println("\nМетод Ритца: \n");  
 y3 = ritzMethod(h);  
 double[] u3 = ritzMethod(h / 100);  
  
 printArrays(y3, u3, h\_diff);  
 }  
  
 public static void main(String[] args){  
 BoundaryValueProblem obj = new BoundaryValueProblem(0.1);  
 obj.solve();  
 }  
}

Вывод программы



Выводы

Данная схема имеет порядок ), т.е. погрешность решения должна быть порядка . При сравнении решений с шагами и (решение с шагом взято в качестве точного), видим, что невязка имеет порядок , что лучше ожидаемой погрешности и в среднем на порядок лучше, чем погрешность решения, полученная с помощью первой схемы, однако в большинстве узлов на порядок меньше, чем погрешность решения, полученного с помощью схемы, построенной методом баланса. Интегралы в коэффициентах данной схемы вычислялись приближенно, используемая квадратурная формула средних прямоугольников обеспечивает требуемый порядок ), однако, поскольку в остатке квадратурной формулы присутствуют производные подынтегральных функций, их значения могут влиять на главный член погрешности решения задачи. Т.к. в схеме, построенной методом баланса, интегралы вычислялись точно, наличие производных подынтегральных функций в главном члене погрешности данной схемы может быть причиной того, что решение, полученное по третьей схеме, имеет невязку на порядок ниже, однако схема все равно имеет порядок аппроксимации, превышающий ожидаемый.

Таким образом, лучшей невязкой обладают результаты, полученные с помощью схемы, построенной методом баланса, однако для ее реализации необходимо точно вычислять интегралы в коэффициентах, что может быть затруднительно. Решение, полученное с помощью схемы, построенной методом Ритца, также имеет невязку порядка при том, что интегралы в коэффициентах вычислялись приближенно. Для всех построенных схем получили невязку, имеющую порядок больший, чем теоретическая погрешность схем.