**Лабораторная работа № 3**

**ОТЧЕТ**

**Тема: «Численное решение смешанной задачи для уравнения теплопроводности»**

Подготовил: студент 3 курса Кафедры БМИ

Кулич С.П.

Минск, 2019

**Постановка задачи:**

На сетке узлов (*h,*) найти численное решение смешанной задачи для одномерного уравнения

теплопроводности с использованием:

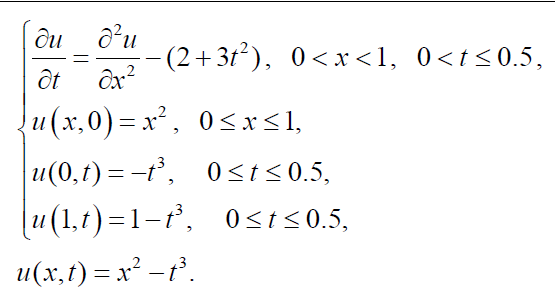
 явной разностной схемы с   *h*  0.1 и *h*  0.1,   *h\*h* / 2 ;

 чисто неявной разностной схемы с   *h*  0.1;

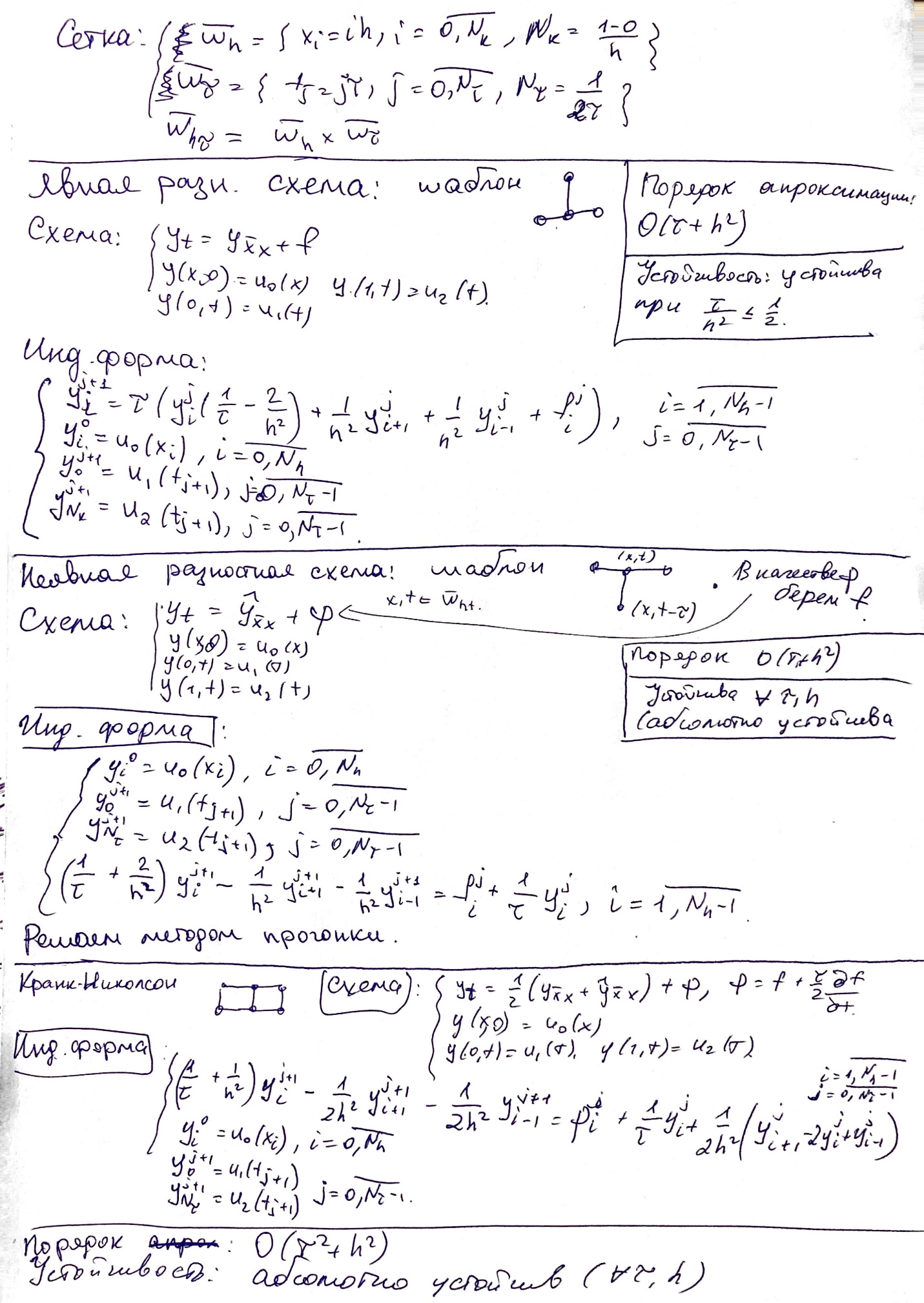
 разностной схемы Кранка-Николсон с   *h*  0.1.

Выписать соответствующие разностные схемы, указать их порядок аппроксимации, указать являются ли схемы абсолютно устойчивыми по начальным данным. Вычислить погрешность численного решения. Построить графики, демонстрирующие устойчивое и неустойчивое поведение явной разностной схемы.

**Условие:**



**Необходимая теория:**



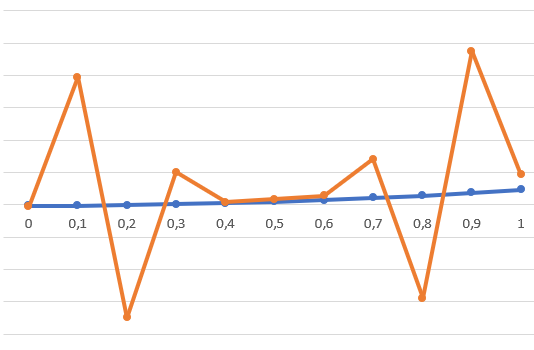
**Результаты:**

Exact error (unstable explicit method): 231,46000

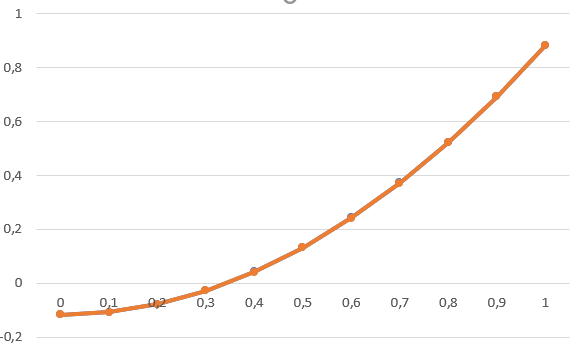
Exact error (stable explicit method): 0,00074 ~ O(h\*h+t)

Exact error (implicit method): 0,01374 ~ O(h\*h+t)

Exact error (Crank-Nicolson method): 0,03215 ~ O(h\*h+t\*t)

**Графики:**

Неустойчивый

****

Устойчивый

**Листинг:**

**App Entry:**

**import** java.util.Map;  
**import** java.util.function.BiFunction;  
**import** java.util.function.Function;  
  
**public class** Main {  
 **private static final** Double ***X\_SIZE*** = 1.0;  
 **private static final** Double ***T\_SIZE*** = 0.5;  
 **private static final** Double ***H*** = 0.1;  
   
 **static** BiFunction<Double, Double, Double> *f* = (x, t) -> -(2 + 3 \* t \* t); *//F* **static** Function<Double, Double> *u0* = x -> x \* x; *//u0* **static** Function<Double, Double> *u1* = t -> -t \* t \* t; *//u1* **static** Function<Double, Double> *u2* = t -> 1 - t \* t \* t; *//u1*

**static** BiFunction<Double, Double, Double> *exactSolution* = (x, t) -> x \* x - t \* t \* t; *//Exact solution* **static** Map.Entry<Double, Double> *dimension* = Map.*entry*(***X\_SIZE***, ***T\_SIZE***); *//dimension* **public static void** main(String[] args) {  
 Double[][] exactSolutionMatrix = *initialiseExactSolutionMatrix*(***H***, ***H***, *dimension*);  
  
 ExplicitMethod firstExplicitMethod = **new** ExplicitMethod(***H***, ***H***, *dimension*, *f*, *u0*, *u1*, *u2*);  
 Double[][] rm1 = firstExplicitMethod.getResult();  
  
 System.***out***.println(String.*format*(**"Exact error (unstable explicit method): %.5f"**, *countError*(rm1, exactSolutionMatrix)));  
  
 Double[][] exactSolutionMatrixSpecial = *initialiseExactSolutionMatrix*(***H***, ***H*** \* ***H*** / 2, *dimension*);  
 ExplicitMethod secondExplicitMethod = **new** ExplicitMethod(***H***, ***H*** \* ***H*** / 2, *dimension*, *f*, *u0*, *u1*, *u2*);  
 Double[][] rm2 = secondExplicitMethod.getResult();  
  
 System.***out***.println(String.*format*(**"Exact error (stable explicit method): %.5f"**, *countError*(rm2, exactSolutionMatrixSpecial)));  
  
 ImplicitMethod implicitMethod = **new** ImplicitMethod(***H***, ***H***, *dimension*, *f*, *u0*, *u1*, *u2*);  
 Double[][] rm3 = implicitMethod.getResult();  
  
 System.***out***.println(String.*format*(**"Exact error (implicit method): %.5f"**, *countError*(rm3, exactSolutionMatrix)));  
  
 CrankNicolsonMethod crankNicolsonMethod = **new** CrankNicolsonMethod(***H***, ***H***, *dimension*, *f*, *u0*, *u1*, *u2*);  
 Double[][] rm4 = crankNicolsonMethod.getResult();  
  
 System.***out***.println(String.*format*(**"Exact error (Crank-Nicolson method): %.5f"**, *countError*(rm4, exactSolutionMatrix)));  
 }  
  
 **private static** Double[][] initialiseExactSolutionMatrix(Double h, Double tau, Map.Entry<Double, Double> dimension) {  
 **int** n1 = (**int**) (dimension.getKey() / h) + 1;  
 **int** n2 = (**int**) (dimension.getValue() / tau) + 1;  
 Double[][] rm = **new** Double[n2][n1];  
  
 **for** (**int** i = 0; i < rm.**length**; i++) {  
 **for** (**int** j = 0; j < rm[i].**length**; j++) {  
 rm[i][j] = *exactSolution*.apply(j \* h, (n2 - 1 - i) \* tau);  
 }  
 }  
 **return** rm;  
 }  
  
 **private static double** countError(Double[][] resultMatrix, Double[][] exactSolutionMatrix) {  
 **double** error = 0d;  
 **for** (**int** i = 0; i < resultMatrix.**length**; i++) {  
 **for** (**int** j = 0; j < resultMatrix[i].**length**; j++) {  
 error = Math.*max*(error, Math.*abs*(resultMatrix[i][j] - exactSolutionMatrix[i][j]));  
 }  
 }  
 **return** error;  
 }  
}

**Explicit method:**

**import** java.util.Map;  
**import** java.util.function.BiFunction;  
**import** java.util.function.Function;  
  
**public class** ExplicitMethod {  
 **final private** Double **h**, **tau**;  
 **final private** Integer **N1**, **N2**;  
 **final private** BiFunction<Double, Double, Double> **f**;  
 **final private** Function<Double, Double> **u0**, **u1**, **u2**;  
  
 *//result matrix* **private** Double[][] **rm**;  
  
 **public** ExplicitMethod(Double h, Double tau, Map.Entry<Double, Double> dimension, BiFunction<Double, Double, Double> f,  
 Function<Double, Double> u0, Function<Double, Double> u1, Function<Double, Double> u2) {  
 **this**.**h** = h;  
 **this**.**tau** = tau;  
 **this**.**f** = f;  
 **this**.**u0** = u0;  
 **this**.**u1** = u1;  
 **this**.**u2** = u2;  
  
 **N1** = (**int**) (dimension.getKey() / h) + 1;  
 **N2** = (**int**) (dimension.getValue() / tau) + 1;  
 **rm** = **new** Double[**N2**][**N1**];  
  
 doAlgorithm();  
 }  
  
 **private void** doAlgorithm() {  
 **for** (**int** i = 0; i < **N1**; i++) {  
 **rm**[**N2** - 1][i] = **u0**.apply(i \* **h**);  
 }  
  
 **for** (**int** i = 1; i < **N2**; i++) {  
 **rm**[**N2** - 1 - i][0] = **u1**.apply(i \* **tau**);  
 **rm**[**N2** - 1 - i][**N1** - 1] = **u2**.apply(i \* **tau**);  
 }  
  
 **double** r = 1 / (**h** \* **h**);  
 **double** k = 1 / **tau** - 2 \* r;  
  
 **for** (**int** i = **N2** - 2; i >= 0; i--) {  
 **for** (**int** j = 1; j < **N1** - 1; j++) {  
 **rm**[i][j] = **tau** \* (k \* **rm**[i + 1][j] + r \* **rm**[i + 1][j + 1] + r \* **rm**[i + 1][j - 1] + **f**.apply(j \* **h**, (**N2** - i - 1) \* **tau**));  
 }  
 }  
 }  
  
 **public** Double[][] getResult() {  
 **return rm**;  
 }  
}

**Implicit method:**

**import** java.util.Map;  
**import** java.util.function.BiFunction;  
**import** java.util.function.Function;  
  
**public class** ImplicitMethod {  
 **final private** Double **h**, **tau**;  
 **final private** Integer **N1**, **N2**;  
 **final private** BiFunction<Double, Double, Double> **f**;  
 **final private** Function<Double, Double> **u0**, **u1**, **u2**;  
  
 *//result matrix* **private** Double[][] **rm**;  
  
 **public** ImplicitMethod(Double h, Double tau, Map.Entry<Double, Double> dimension, BiFunction<Double, Double, Double> f,  
 Function<Double, Double> u0, Function<Double, Double> u1, Function<Double, Double> u2) {  
 **this**.**h** = h;  
 **this**.**tau** = tau;  
 **this**.**f** = f;  
 **this**.**u0** = u0;  
 **this**.**u1** = u1;  
 **this**.**u2** = u2;  
  
 **N1** = (**int**) (dimension.getKey() / h) + 1;  
 **N2** = (**int**) (dimension.getValue() / tau) + 1;  
 **rm** = **new** Double[**N2**][**N1**];  
  
 doAlgorithm();  
 }  
  
 **private void** doAlgorithm() {  
 **for** (**int** i = 0; i < **N1**; i++) {  
 **rm**[**N2** - 1][i] = **u0**.apply(i \* **h**);  
 }  
  
 **for** (**int** i = 1; i < **N2**; i++) {  
 **rm**[**N2** - 1 - i][0] = **u1**.apply(i \* **tau**);  
 **rm**[**N2** - 1 - i][**N1** - 1] = **u2**.apply(i \* **tau**);  
 }  
  
 **for** (**int** i = **N2** - 2; i >= 0; i--) {  
 **rm**[i] = doDiagonalMatrixAlgorithm(i);  
 }  
 }  
  
 **private** Double[] doDiagonalMatrixAlgorithm(**int** i) {  
 Double[][] A = **new** Double[**N1**][**N1**];  
 Double[] B = **new** Double[**N1**];  
  
 A[0][0] = 1.0;  
 A[0][1] = 0.0;  
 B[0] = **rm**[i][0];  
  
 A[**N1** - 1][**N1** - 1] = 1.0;  
 A[**N1** - 1][**N1** - 2] = 0.0;  
 B[**N1** - 1] = **rm**[i][**N1** - 1];  
  
 **double** r = 1 / (**h** \* **h**);  
 **double** k = 1 / **tau** + 2 \* r;  
 **for** (**int** j = 1; j < **N1** - 1; j++) {  
 A[j][j - 1] = -r;  
 A[j][j] = k;  
 A[j][j + 1] = -r;  
  
 B[j] = **f**.apply(j \* **h**, (**N2** - i - 1) \* **tau**) + **rm**[i + 1][j]/**tau**;  
 }  
 **final** TridiagonalMatrixAlgorithm result = **new** TridiagonalMatrixAlgorithm(A, B);  
 **return** result.getResult();  
 }  
  
 **public** Double[][] getResult() {  
 **return rm**;  
 }  
}

**Crunke-Nicolson method:**

**import** java.util.Map;  
**import** java.util.function.BiFunction;  
**import** java.util.function.Function;  
  
**public class** CrankNicolsonMethod {  
 **final private** Double **h**, **tau**;  
 **final private** Integer **N1**, **N2**;  
 **final private** BiFunction<Double, Double, Double> **f**;  
 **final private** Function<Double, Double> **u0**, **u1**, **u2**;  
  
 *//result matrix* **private** Double[][] **rm**;  
  
 **public** CrankNicolsonMethod(Double h, Double tau, Map.Entry<Double, Double> dimension, BiFunction<Double, Double, Double> f,  
 Function<Double, Double> u0, Function<Double, Double> u1, Function<Double, Double> u2) {  
 **this**.**h** = h;  
 **this**.**tau** = tau;  
 **this**.**f** = f;  
 **this**.**u0** = u0;  
 **this**.**u1** = u1;  
 **this**.**u2** = u2;  
  
 **N1** = (**int**) (dimension.getKey() / h) + 1;  
 **N2** = (**int**) (dimension.getValue() / tau) + 1;  
 **rm** = **new** Double[**N2**][**N1**];  
  
 doAlgorithm();  
 }  
  
 **private void** doAlgorithm() {  
 **for** (**int** i = 0; i < **N1**; i++) {  
 **rm**[**N2** - 1][i] = **u0**.apply(i \* **h**);  
 }  
  
 **for** (**int** i = 1; i < **N2**; i++) {  
 **rm**[**N2** - 1 - i][0] = **u1**.apply(i \* **tau**);  
 **rm**[**N2** - 1 - i][**N1** - 1] = **u2**.apply(i \* **tau**);  
 }  
  
 **for** (**int** i = **N2** - 2; i >= 0; i--) {  
 **rm**[i] = doDiagonalMatrixAlgorithm(i);  
 }  
 }  
  
 **private** Double[] doDiagonalMatrixAlgorithm(**int** i) {  
 Double[][] A = **new** Double[**N1**][**N1**];  
 Double[] B = **new** Double[**N1**];  
  
 A[0][0] = 1.0;  
 A[0][1] = 0.0;  
 B[0] = **rm**[i][0];  
  
 A[**N1** - 1][**N1** - 1] = 1.0;  
 A[**N1** - 1][**N1** - 2] = 0.0;  
 B[**N1** - 1] = **rm**[i][**N1** - 1];  
  
 **double** r = 1 / (**h** \* **h**);  
 **double** k = 1 / **tau** + r;  
 **for** (**int** j = 1; j < **N1** - 1; j++) {  
 A[j][j - 1] = -r / 2;  
 A[j][j] = k;  
 A[j][j + 1] = -r / 2;  
  
 B[j] = **f**.apply(j \* **h**, (**N2** - i - 1) \* **tau**) + **tau** / 2 \* (-6 \* (**N2** - i - 1) \* **tau**) + **rm**[i + 1][j] / **tau** +  
 r / 2 \* (**rm**[i + 1][j + 1] - 2 \* **rm**[i + 1][j] + **rm**[i + 1][j - 1]);  
 }  
 **final** TridiagonalMatrixAlgorithm result = **new** TridiagonalMatrixAlgorithm(A, B);  
 **return** result.getResult();  
 }  
  
 **public** Double[][] getResult() {  
 **return rm**;  
 }  
}

**Three diagonal matrix method:**

**public class** TridiagonalMatrixAlgorithm {  
 **private final** Double[][] **A**;  
 **private final** Double[] **B**;  
 **private** Double[] **result**;  
 **private final int size**;  
  
 **public** TridiagonalMatrixAlgorithm(Double[][] A, Double[] B) {  
 **this**.**A** = A;  
 **this**.**B** = B;  
 **size** = A[0].**length**;  
 **result** = **new** Double[**size**];  
  
 doAlgorithm();  
 }  
  
 **private void** doAlgorithm() {  
 Double[] alpha = **new** Double[**size**];  
 Double[] betta = **new** Double[**size**];  
 **double** z;  
  
 alpha[1] = **A**[0][1] / **A**[0][0];  
 betta[1] = **B**[0] / **A**[0][0];  
  
 **for** (**int** i = 1; i < **size** - 1; i++) {  
 z = **A**[i][i] + **A**[i][i - 1] \* alpha[i];  
 alpha[i + 1] = -**A**[i][i + 1] / z;  
 betta[i + 1] = (**B**[i] - **A**[i][i - 1] \* betta[i]) / z;  
 }  
  
 **result**[**size** - 1] = (**B**[**size** - 1] - **A**[**size** - 1][**size** - 2] \* betta[**size** - 1])  
 / (**A**[**size** - 1][**size** - 1] + **A**[**size** - 1][**size** - 2] \* alpha[**size** - 1]);  
  
 **for** (**int** i = **size** - 2; i >= 0; i--) {  
 **result**[i] = alpha[i + 1] \* **result**[i + 1] + betta[i + 1];  
 }  
 }  
  
 **public** Double[] getResult() {  
 **return result**;  
 }  
}