



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования

«Московский государственный технический университет имени Н.Э.  
Баумана (национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Робототехника и комплексная автоматизация  
КАФЕДРА Системы автоматизированного проектирования (САПР)

## Отчет о выполнении домашней работы по дисциплине "Модели и методы анализа проектных решений"

Студент:	Дунайцев Александр Иванович
Группа:	РК6-64Б
Тип задания:	Вариант 67
Тема:	Метод конечных разностей при решении задач теплопроводности

Студент

\_\_\_\_\_  
подпись, дата

Дунайцев А. И

Преподаватель

\_\_\_\_\_  
подпись, дата

Трудоношин В. А.

Москва, 2022

# Содержание

<b>1</b>	<b>Задание</b>	<b>3</b>
<b>2</b>	<b>Теоретическая часть</b>	<b>3</b>
2.1	Нестационарное уравнение теплопроводности . . . . .	3
2.2	Метод конечных разностей . . . . .	4
<b>3</b>	<b>Описание работы программы</b>	<b>4</b>
<b>4</b>	<b>Решение задачи</b>	<b>5</b>
4.1	Решение задачи с помощью разработанной программы . . . . .	5
4.2	Решение задачи с помощью пакета ANSYS . . . . .	6
4.3	Сравнение результатов работы программы . . . . .	6
<b>5</b>	<b>Код программы</b>	<b>6</b>
5.1	Matrxix . . . . .	6
5.2	CalculationUtils . . . . .	10
5.3	SolutionStorage . . . . .	12
5.4	Model . . . . .	15
5.5	Решение задачи при помощи реализованных библиотек . . . . .	22

# 1 Задание

С помощью явной разностной схемы решить нестационарное уравнение теплопроводности для трубы, изображенной на рис. 1, там же указаны размеры сторон.

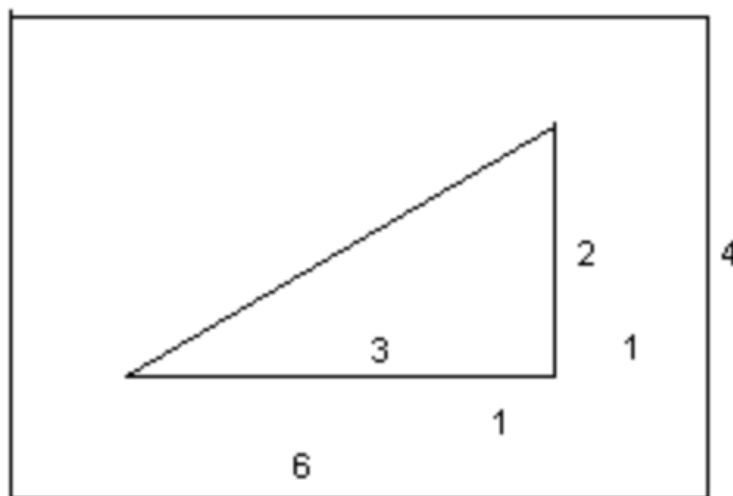


Рис. 1: Форма трубы

Граничные условия: Внутри трубы задано условие:  $\frac{\partial T}{\partial n} = T$ .

На внешних границах заданы следующие условия: верхняя сторона - 20, на остальных выполняется условие  $\frac{\partial T}{\partial n} = 40$ .

Начальное значение температуры трубы - 20 градусов.

При выводе результатов показать динамику изменения температуры (например, с помощью цветовой гаммы).

## 2 Теоретическая часть

### 2.1 Нестационарное уравнение теплопроводности

В отличие от стационарных задач, при постановке нестационарных задач нас интересует определение состояния сплошной среды переменное во времени. Для решения подобного рода задач определяют краевые условия, то есть совокупность граничных условий и условий состояния среды в начальный момент времени (начальных условий).

В контексте поставленной задачи необходимо найти значения температуры всех точек среды (в данном случае трубы), в разные моменты времени. Совокупность значений температуры во всех точках среды в определенный момент времени называют температурным полем.

Для решения двумерных задач, связанных с поиском значений температурного поля в различные моменты времени, необходимо иметь дифференциальное уравнение теплопроводности пластины, которое связывает температуру пластины, время и пространственными координатами исследуемой среды. В декартовой системе координат такое уравнение имеет следующий вид:

$$\frac{\partial T}{\partial t} = \frac{\lambda}{c\rho} \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) + \frac{w}{c\rho},$$

где  $\lambda$  - коэффициент теплопроводности,  $c$  - теплоемкость,  $p$  - плотность,  $w$  - мощность тепловыделения.

## 2.2 Метод конечных разностей

Для решения поставленной задачи воспользуемся методом конечных разностей (далее МКР).

Смысл этого метода заключается в представлении исходного объекта его математическим аналогом, который представляет собой сетку, в узлах которой находятся исследуемые значения. Каждый узел соответствует значению в определенной пространственной координате исходного объекта, таким образом координата узла вычисляется следующим  $(x_i, y_j) = (i\Delta x, j\Delta y)$ , где  $i \in [0, \dots, n]$ ,  $j \in [0, \dots, m]$ , а  $n, m$  - количество узлов вдоль оси абсцисс и оси ординат соответственно.

Далее можно перейти к разностным аналогам частных производных в двумерном пространстве. Записать их можно следующим образом: Первая производная

$$\frac{\partial T}{\partial y} = \frac{T_{i,j+1}^k - T_{i,j}^k}{\Delta y}$$

Вторая производная

$$\frac{\partial^2 T}{\partial y^2} = \frac{T_{i,j+1}^k - 2T_{i,j}^k + T_{i,j-1}^k}{\Delta y^2}.$$

Здесь  $k$  - это индекс узла по времени,  $i$  - индекс узла по координате  $x$ ,  $j$  - индекс по координате  $y$ . Важно отметить, что в данном примере записи разностной формы производная берется по координате  $y$ . Однако в зависимости от переменной, по которой берется производная, будет изменяться индекс в разностной схеме, соответствующий этой переменной.

Таким образом, можно записать уравнение теплопроводности с помощью явной разностной схемы следующим образом:

$$\frac{T_{i,j}^{k+1} - T_{i,j}^k}{\Delta t} = \frac{\lambda}{cp} \left( \frac{T_{i+1,j}^k - 2T_{i,j}^k + T_{i-1,j}^k}{\Delta x^2} + \frac{T_{i,j+1}^k - 2T_{i,j}^k + T_{i,j-1}^k}{\Delta y^2} \right) + \frac{w}{cp}.$$

Так как исследуемый объект имеет отверстие в центре неправильной формы, имеет смысл брать сетку таким образом, чтобы ее узлы оказывались на границах треугольного отверстия в центре трубый.

## 3 Описание работы программы

Решение нестационарного уравнения теплопроводности сводится к тому, чтобы на всех временных слоях получить значения во всех узлах сетки заданной модели.

Чтобы задать сетку, довольно удобно воспользоваться двумерным массивом, который учитывает форму исследуемого объекта. Таким образом, каждая ячейка массива будет хранить в себе значение узла. Однако, поскольку хранение двумерного массива большого в памяти компьютера довольно накладно по скорости доступа к элементу массива, следует заменить двумерный массив на одномерный, класс которого будет иметь семантику работы двумерного массива.

После создания сетки, решение поставленной задачи сводится к итерационному вычислению значений всех узлов сетки на всех временных слоях. Следует отметить, что при этом учитываются граничные условия заданные на исследуемый объект.

Решение нестационарных задач теплопроводности может оказаться достаточно затратным по памяти, при условии, что заданы достаточно маленькие приращения по пространственным координатам и большое количество временных слоев. В связи с этим программная реализация должна сводиться только к расчетам на всех временных слоях, а не попыткам сохранить их в оперативной памяти компьютера, поэтому реализованный класс модели не отвечает за хранение вычислений. Он лишь принимает полиморфный объект хранилища, который отвечает за сохранение результата расчетов, если это необходимо.

## 4 Решение задачи

### 4.1 Решение задачи с помощью разработанной программы

Для более наглядного отображения работы программы, было принято решение задать температуру внутри трубы равной 0 градусам. Однако, следует отметить, что значения внутри трубы никак не учитываются и не влияют на граничные условия, заданные на границах отверстия. В результате интегрирования по времени в течении 25 секунд, получилась тепловая карта, представленная на рисунке 3.

146.993	126.993	103.96	75.9732	20	20	20	20	20	20	20	20	40
126.993	107.072	84.0267	56.0178	0	0	0	0	0	0	0	0	20
112.000	92.0033	70.5678	47.0452	0	0	0	0	0	0	0	0	20
100.000	80.3746	60.531	40.354	0	0	0	0	0	0	0	0	20
90.008	70.0613	52.0755	34.939	16.6554	11.1036	0	0	0	0	0	0	20
78.9532	58.9911	43.7958	31.3868	20.9235	14.8128	9.87521	0	0	0	0	0	20
63.982	44.0017	33.4641	26.436	21.1836	17.5725	14.4225	10.3193	6.87953	0	0	0	20
20	20	20	20	20	20	20	20	20	20	20	20	20

Рис. 2: Результат работы программы в табличном виде с шагом  $h = 0.5$

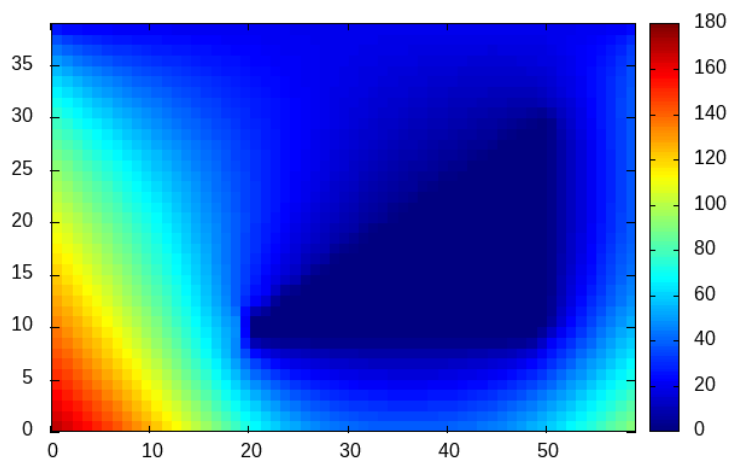


Рис. 3: Графическое изображение результатов расчетов с шагом 0.1

## 4.2 Решение задачи с помощью пакета ANSYS

Для решения задачи теплопроводности трубы средствами пакета ANSYS, была построена модель трубы, проведено разбиение на конечные элементы, наложены граничные условия. Визуализация распределения температур для момента времени  $t = 25$  сек представлена на рисунке 4.

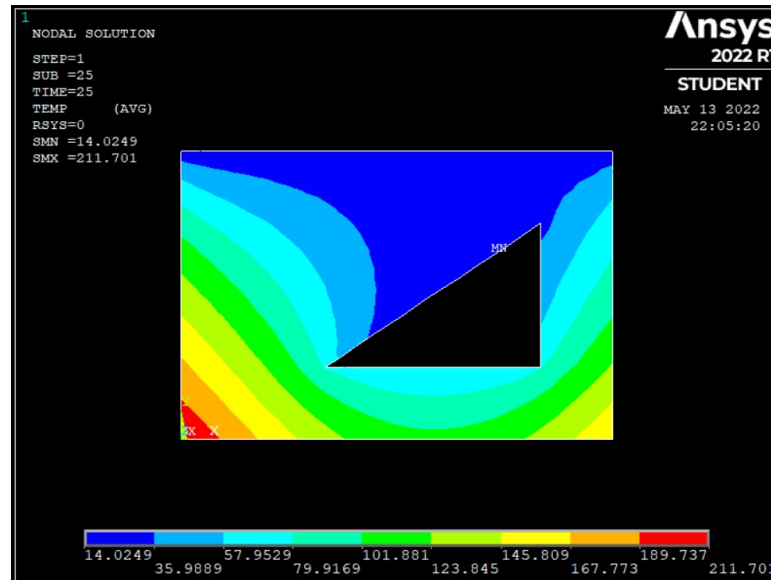


Рис. 4: Графическое изображение результатов расчетов с помощью ANSYS

## 4.3 Сравнение результатов работы программы

На основе полученных распределений температур, представленных на рисунке 4 и рисунке 2, можно сделать вывод, что решение задачи при помощи пакета ANSYS и при помощи написанной программы совпадает. Отсюда можно заключить, что программа, реализованная в рамках данной работы, работает корректно.

# 5 Код программы

## 5.1 Matrxix

```
1 #ifndef FINITEDIFFERENCEMETHOD_MATRIX_HPP_
2 #define FINITEDIFFERENCEMETHOD_MATRIX_HPP_
3
4 #include <algorithm>
5 #include <array>
6 #include <cstdint>
7 #include <exception>
8 #include <memory>
9 #include <type_traits>
10 #include <vector>
11
12 namespace mtrx {
13 namespace exceptions {
14 class BaseMatrixException : public std::exception {
15     [[nodiscard]] const char *what() const noexcept override {
```

```

16     return "Matrix error occur";
17 }
18 };
19
20 class MatrixSizeException : public BaseMatrixException {
21     [[nodiscard]] const char *what() const noexcept override {
22         return "Matrix size error occur. You may have gone beyond the matrix.";
23     }
24 };
25 } // namespace exceptions
26
27 /**
28  * I do not really want to complicate the code in the context of my tasks.
29  * Therefore, I add the implementation of the store and accessor as much as
30  * possible. But MatrixBase provide the simple interface, that using in
31  * FDM implementation.
32  *
33  * @note All derived classes have to necessary define Type
34  *
35  * @tparam Tp matrix elements type
36  */
37 namespace base {
38     template <typename Tp>
39     class MatrixBase {
40     public:
41         using Type = Tp;
42         virtual void SetValue(size_t row, size_t col, Tp &&val) = 0;
43         virtual const Tp &GetValue(size_t row, size_t col) const = 0;
44         virtual size_t SizeRows() const = 0;
45         virtual size_t SizeCols() const = 0;
46         [[maybe_unused]] virtual void FillMatrix([[maybe_unused]] Tp val) {}
47
48         virtual ~MatrixBase() = default;
49     };
50
51     template <typename Tp>
52     class MatrixDynamicBase : public MatrixBase<Tp> {
53     public:
54         using Type = Tp;
55         virtual void SetRows(size_t rows) = 0;
56         virtual void SetCols(size_t cols) = 0;
57         void SetSize(size_t rows, size_t cols) {
58             this->SetRows(rows);
59             this->SetCols(cols);
60         }
61
62         virtual ~MatrixDynamicBase() = default;
63     };
64 } // namespace base
65
66 /**
67  * Simplest matrix implementation. Matrix class in this program maintains
68  * only basic operations and using as structure for easy to store and
69  * access values.
70  *
71  * @tparam Tp matrix elements type
72  * @tparam Cols amount of columns
73  * @tparam Rows amount of m_rows
74  */

```

```

75 template <typename Tp, size_t Rows, size_t Cols>
76 class Matrix : public base::MatrixBase<Tp> {
77 public:
78     using Type = Tp;
79     using MatrixStorageType = std::array<Type, Cols * Rows>;
80     static constexpr auto MatrixAccessor = [](size_t row, size_t col) ->
        size_t {
81         return row * Cols + col;
82     };
83
84     void SetValue(size_t row, size_t col, Tp &&val) override;
85     const Tp &GetValue(size_t row, size_t col) const override;
86     [[nodiscard]] virtual size_t SizeRows() const override { return Rows; }
87     [[nodiscard]] virtual size_t SizeCols() const override { return Cols; }
88
89     Matrix() = default;
90
91 private:
92     MatrixStorageType m_storage;
93
94     [[nodiscard]] bool CheckAccess(size_t row, size_t col) const {
95         return col < Cols && row < Rows;
96     }
97 };
98
99 template <typename Tp, size_t Rows, size_t Cols>
100 void Matrix<Tp, Rows, Cols>::SetValue(size_t row, size_t col, Tp &&val) {
101     if (!CheckAccess(row, col)) {
102     }
103     m_storage[MatrixAccessor(row, col)] = std::forward<Tp>(val);
104 }
105
106 template <typename Tp, size_t Rows, size_t Cols>
107 const Tp &Matrix<Tp, Rows, Cols>::GetValue(size_t row, size_t col) const {
108     if (!CheckAccess(row, col)) {
109     }
110     return m_storage[MatrixAccessor(row, col)];
111 }
112
113 /**
114  * This matrix builder is created only because I started implemented
115  * current library with only static matrix
116  */
117 template <typename MatrixClass>
118 class MatrixCreatorStatic {
119 public:
120     using BaseType = base::MatrixBase<typename MatrixClass::Type>;
121     using MatrixPointer = std::unique_ptr<BaseType>;
122
123     MatrixPointer operator()() {
124         static_assert(std::is_base_of_v<BaseType, MatrixClass>);
125         return std::make_unique<MatrixClass>();
126     }
127 };
128
129 template <typename Tp>
130 class MatrixDynamic : public base::MatrixDynamicBase<Tp> {
131 public:
132     using Type = Tp;

```



```

133 using MatrixStorageType = std::vector<Type>;
134
135 void SetRows(size_t rows) override {
136     m_rows = rows;
137     m_storage.resize(m_rows * m_cols);
138 }
139 void SetCols(size_t cols) override {
140     m_cols = cols;
141     m_storage.resize(m_rows * m_cols);
142 }
143 void SetSize(size_t rows, size_t cols) {
144     m_rows = rows;
145     m_cols = cols;
146     m_storage.resize(rows * cols);
147 }
148 [[nodiscard]] size_t SizeRows() const override { return m_rows; }
149 [[nodiscard]] size_t SizeCols() const override { return m_cols; }
150
151 void SetValue(size_t row, size_t col, Tp &&val) override;
152 const Tp &GetValue(size_t row, size_t col) const override;
153 void FillMatrix(Tp val) override;
154
155 MatrixDynamic() = default;
156 MatrixDynamic(size_t rows, size_t cols) : m_cols(m_cols), m_rows(m_rows) {
157     SetSize(rows, cols);
158 }
159
160 private:
161     MatrixStorageType m_storage;
162     size_t m_cols;
163     size_t m_rows;
164
165     [[nodiscard]] bool CheckAccess(size_t row, size_t col) const {
166         return col < m_cols && row < m_rows;
167     }
168
169     [[nodiscard]] size_t MatrixAccessor(size_t row, size_t col) const {
170         return row * m_cols + col;
171     }
172 };
173
174 template <typename Tp>
175 void MatrixDynamic<Tp>::SetValue(size_t row, size_t col, Tp &&val) {
176     if (!CheckAccess(row, col)) {
177         throw exceptions::MatrixSizeException();
178     }
179     m_storage[MatrixAccessor(row, col)] = std::forward<Tp>(val);
180 }
181
182 template <typename Tp>
183 const Tp &MatrixDynamic<Tp>::GetValue(size_t row, size_t col) const {
184     if (!CheckAccess(row, col)) {
185         throw exceptions::MatrixSizeException();
186     }
187     return m_storage[MatrixAccessor(row, col)];
188 }
189
190 template <typename Tp>
191 void MatrixDynamic<Tp>::FillMatrix(Tp val) {

```

```

192     std::fill(m_storage.begin(), m_storage.end(), val);
193 }
194
195 class MatrixCreatorDynamic {
196 public:
197     template <typename Tp>
198     using TargetType = MatrixDynamic<Tp>;
199     template <typename Tp>
200     using BaseType = base::MatrixDynamicBase<Tp>;
201     template <typename Tp>
202     using Pointer = std::shared_ptr<BaseType<Tp>>;
203
204     template <typename Tp>
205     Pointer<Tp> Build() {
206         return std::forward<Pointer<Tp>>(std::make_unique<TargetType<Tp>>());
207     }
208 };
209 } // namespace mtrx
210
211 #endif // FINITEDIFFERENCEMETHOD_MATRIX_HPP_

```

Листинг 1: Matrix.hpp

## 5.2 CalculationUtils

```

1 #ifndef FINITEDIFFERENCEMETHOD_CALCULATIONUTILS_HPP_
2 #define FINITEDIFFERENCEMETHOD_CALCULATIONUTILS_HPP_
3
4 #include <memory>
5 #include <tuple>
6
7 namespace fdm {
8 /**
9  * I decided to make the restrictions separate from the model class,
10  * since if desired, these classes can be reused for other specific models.
11  */
12 namespace restr {
13     template <typename ModelNodeType>
14     class BaseRestriction {
15     public:
16         virtual ModelNodeType operator()(ModelNodeType, double delta) = 0;
17         virtual ~BaseRestriction() = default;
18     };
19
20 /**
21  * As we know, first kind restriction return constant value not depend
22  * on boundary derivative value.
23  */
24     template <typename ModelNodeType>
25     class FirstKindRestriction : public BaseRestriction<ModelNodeType> {
26     public:
27         explicit FirstKindRestriction(double constant = 0.0) : m_constant(constant) {}
28         ModelNodeType operator()(ModelNodeType inner, double delta) override {
29             return m_constant;
30         }
31
32     private:
33         double m_constant;
34     };
35 }

```

```

34 };
35
36 template <typename ModelNodeType>
37 class SecondKindRestriction : public BaseRestriction<ModelNodeType> {
38 public:
39     explicit SecondKindRestriction(double constant = 0.0)
40         : m_constant(constant) {}
41     ModelNodeType operator()(ModelNodeType inner, double delta) override {
42         return inner + m_constant * delta;
43     }
44
45 private:
46     double m_constant;
47 };
48
49 template <typename ModelNodeType>
50 class ThirdKindRestriction : public BaseRestriction<ModelNodeType> {
51 public:
52     ModelNodeType operator()(ModelNodeType inner, double delta) override {
53         return inner / (1 + delta);
54     }
55 };
56
57 /*
58  * Restriction is the callable object, that calculate boundary values.
59  * Parameters are passed to the function, in the bellow signature,
60  * in order of taking the derivative from inside to outside. Last parameter
61  * represent differential specific derivative increment value.
62  */
63 template <typename ModelNodeType>
64 using BoundaryRestrincionType = BaseRestriction<ModelNodeType>;
65 template <typename ModelNodeType>
66 using BoundaryRestrincionPointerType =
67     std::shared_ptr<BoundaryRestrincionType<ModelNodeType>>;
68 // Restrictions on the outer boundaries of tube.
69 template <typename ModelNodeType>
70 using BoundaryRestrictionsStorageType =
71     std::array<BoundaryRestrincionPointerType<ModelNodeType>, 4>;
72 // Order of restrictions
73 constexpr size_t UP_RESTRICTION = 0;
74 constexpr size_t DOWN_RESTRICTION = 1;
75 constexpr size_t LEFT_RESTRICTION = 2;
76 constexpr size_t RIGHT_RESTRICTION = 3;
77 } // namespace restr
78
79 namespace equations {
80 /**
81  * @param params pass tuple of Ti,j, Ti-1,j, Ti+1,j, Ti,j-1, Ti,j+1,
82  * delta
83  * t, delta x, delta y and a
84  */
85 template <typename ModelNodeType>
86 using HeatConductionParamsType =
87     std::tuple<ModelNodeType, ModelNodeType, ModelNodeType, ModelNodeType,
88         ModelNodeType, double, double, double, double>;
89 /**
90  * Solve the non-stationary problem of heat conduction by an explicit method
91  *
92  * @tparam ModelNodeType model node type

```

```

91  * @param params pass tuple of Ti,j, Ti-1,j, Ti+1,j, Ti,j-1, Ti,j+1,
    delta
92  * t, delta x, delta y and a
93  * @return predict node Ti,j value on new time layer
94  */
95  template <typename ModelNodeType>
96  ModelNodeType HeatConductionProblem(
97      const HeatConductionParamsType<ModelNodeType> &params) {
98      auto [T_curr, T_x_past, T_x_next, T_y_past, T_y_next, dt, dx, dy, a] =
        params;
99      double dx2 = dx * dx;
100     double dy2 = dy * dy;
101     ModelNodeType Dx = (T_x_next - 2 * T_curr + T_x_past) / dx2;
102     ModelNodeType Dy = (T_y_next - 2 * T_curr + T_y_past) / dy2;
103     ModelNodeType D = Dx + Dy;
104     return a * dt * D + T_curr;
105 }
106 } // namespace equations
107
108 } // namespace fdm
109
110 #endif // FINITEDIFFERENCEMETHOD_CALCULATIONUTILS_HPP_

```

Листинг 2: CalculationUtils.hpp

### 5.3 SolutionStorage

```

1  #ifndef FINITEDIFFERENCEMETHOD_SOLUTIONSTORAGE_HPP_
2  #define FINITEDIFFERENCEMETHOD_SOLUTIONSTORAGE_HPP_
3
4  #include <exception>
5  #include <fstream>
6  #include <iomanip>
7  #include <iostream>
8  #include <memory>
9  #include <string_view>
10
11 #include "Matrix.hpp"
12
13 namespace fdm {
14     namespace solution {
15         namespace exceptions {
16             class SolutionStorageException : public std::exception {
17             public:
18                 [[nodiscard]] const char *what() const noexcept override {
19                     return "Storage exception occur";
20                 }
21             };
22
23             class FileNotOpenException : public std::exception {
24             public:
25                 [[nodiscard]] const char *what() const noexcept override {
26                     return "One of files (configuration or plot) was not open";
27                 }
28             };
29         } // namespace exceptions
30     } // namespace fdm
31     /**
32     * Basic model's mesh storing class. Class provides functions, that helps

```

```

33 * to manage mesh optimal storage. This class is necessary to use if you
34 * dont want to lose your calculations data, because Model class don't
35 * store all time integrated layers.
36 *
37 * @tparam MeshNodesType specify mesh nodes type
38 */
39 template <typename MeshNodesType>
40 class SolutionStorageBase {
41 public:
42     using MeshType = mtrx::base::MatrixBase<MeshNodesType>;
43     using MeshPointerType = std::shared_ptr<MeshType>;
44
45     virtual void CommitLayer(const MeshPointerType &mesh_ptr) = 0;
46     virtual ~SolutionStorageBase() = default;
47 };
48
49 /**
50 * Simplest storage, that commit all integrated layers of
51 * model in standard output
52 */
53 template <typename MeshNodesType>
54 class StandardStreamStorage : public SolutionStorageBase<MeshNodesType> {
55 public:
56     std::ostream &StandardStreamDefinition = std::cout;
57
58     void CommitLayer(
59         const typename SolutionStorageBase<MeshNodesType>::MeshPointerType
60         &mesh_ptr) override {
61         // StandardStreamDefinition.precision(2);
62         for (size_t i = 0; i < mesh_ptr->SizeRows(); ++i) {
63             for (size_t j = 0; j < mesh_ptr->SizeCols(); ++j) {
64                 StandardStreamDefinition << mesh_ptr->GetValue(i, j) << ' ';
65             }
66             StandardStreamDefinition << '\n';
67         }
68         StandardStreamDefinition << '\n';
69     }
70 };
71
72 template <typename MeshNodesType>
73 class StaticGnuplotHeatmapStorage : public SolutionStorageBase<MeshNodesType>
74 {
75 public:
76     using StringType = std::string;
77
78     explicit StaticGnuplotHeatmapStorage(
79         const StringType &file_prefix)
80         : m_file_output_prefix(file_prefix) {
81         m_plot_file_name =
82             std::string(m_file_output_prefix) + std::string(plot_extension);
83         m_config_file_name =
84             std::string(m_file_output_prefix) + std::string(config_extension);
85
86         m_plot_output.open(m_plot_file_name.data());
87         m_config_output.open(m_config_file_name.data());
88         if (!m_plot_output.is_open() || !m_config_output.is_open()) {
89             throw exceptions::FileNotOpenException();
90         }
91         WriteConfig();
92     }

```

```

91 }
92
93 void CommitLayer(
94     const typename SolutionStorageBase<MeshNodesType>::MeshPointerType
95     &mesh_ptr) override {
96     std::ostream &StandardStreamDefinition = m_plot_output;
97     for (size_t i = 0; i < mesh_ptr->SizeRows(); ++i) {
98         for (size_t j = 0; j < mesh_ptr->SizeCols(); ++j) {
99             StandardStreamDefinition << std::setw(8) << mesh_ptr->GetValue(i, j)
100             << ' ';
101         }
102         StandardStreamDefinition << '\n';
103     }
104     StandardStreamDefinition << '\n';
105 }
106
107 ~StaticGnuplotHeatmapStorage() override {
108     m_plot_output.close();
109     m_config_output.close();
110 }
111
112 private:
113     StringType m_file_output_prefix;
114     StringType m_plot_file_name;
115     StringType m_config_file_name;
116     std::ofstream m_config_output;
117     std::ofstream m_plot_output;
118
119     StringType plot_extension = ".plt";
120     StringType config_extension = ".cfg";
121     StringType mapping_extension = ".png";
122     StringType default_file_prefix = "def_output";
123
124 void WriteConfig() {
125     StringType terminal_def{"set terminal png\n"};
126     StringType output_file_def{
127         "set output '" + std::string(m_file_output_prefix) +
128         std::string(mapping_extension) + "'\n"};
129     StringType scale_def{
130         "set autoscale yfix\nset autoscale xfix\n"};
131     StringType palette_def{
132         "set palette defined (0 0 0 0.5, 1 0 0 1, 2 0 0.5 1, 3 0 1 1, 4 0.5
133         1 "
134         "0.5, 5 1 1 0, 6 1 0.5 0, 7 1 0 0, 8 0.5 0 0)\n"};
135     StringType map_type_def{"set pm3d map\n"};
136     StringType splot_def{
137         "splot '" + std::string(m_plot_file_name) + "' matrix notitle\n"};
138     m_config_output << terminal_def;
139     m_config_output << output_file_def.data();
140     m_config_output << scale_def;
141     m_config_output << palette_def;
142     m_config_output << map_type_def;
143     m_config_output << splot_def.data();
144 }
145 };
146
147 template <typename MeshNodesType>
148 class PlaceholderStorage : public SolutionStorageBase<MeshNodesType> {
149 public:

```

```

148 void CommitLayer(
149     const typename SolutionStorageBase<MeshNodesType>::MeshPointerType
150     &mesh_ptr) override {}
151 };
152 } // namespace solution
153 } // namespace fdm
154
155 #endif // FINITEDIFFERENCEMETHOD_SOLUTIONSTORAGE_HPP_

```

Листинг 3: SolutionStorage.hpp

## 5.4 Model

```

1 #ifndef FINITEDIFFERENCEMETHOD_MODEL_HPP_
2 #define FINITEDIFFERENCEMETHOD_MODEL_HPP_
3
4 #include <array>
5 #include <exception>
6 #include <fstream>
7 #include <functional>
8 #include <iostream>
9 #include <tuple>
10 #include <utility>
11 #include <vector>
12
13 #include "CalculationUtils.hpp"
14 #include "Matrix.hpp"
15 #include "SolutionStorage.hpp"
16
17 // As you'll see, I'm a big fan of readable aliases. Don't swear if this
18 // makes
19 // my code unreadable. =)
20 namespace fdm {
21 /**
22  * Since finite difference coding in general cannot be done for
23  * free-form geometry, I wrote a model that solves the
24  * non-stationary heat conduction problem in an explicit way on
25  * the geometry of a tube with a triangular hole.
26  */
27 class Model {
28 public:
29     /**
30      * If you wish to change the implementation of the class with
31      * your own data types that implement the interface declared
32      * in the Matrix.h file, change the aliases declared here.
33      * This is done so as not to complicate the understanding of
34      * the program code.
35      */
36     using ModelNodeType = double;
37     constexpr static ModelNodeType DefModelVal = 0.0;
38     using MatrixBuilder = mtrx::MatrixCreatorDynamic;
39     using MatrixPointerType = MatrixBuilder::Pointer<ModelNodeType>;
40
41     /**
42      * Actually I could use std::pair, but in my opinion Point
43      * is more representative.
44      */
45     struct Point {

```

```

46 ModelNodeType x;
47 ModelNodeType y;
48 Point() : x(DefModelVal), y(DefModelVal) {}
49 Point(ModelNodeType _x, ModelNodeType _y) : x(_x), y(_y) {}
50 };
51
52 /*
53  * Triangular hole in the center of tube. I think
54  * good idea to store it in fixed size array.
55  */
56 using HoleGeometry = std::array<Point, 3>;
57
58 // Finally, after all this NECESSARY definitions - code!!!
59 Model()
60     : m_mesh_ptr_present(),
61       m_mesh_ptr_last(),
62       m_width(DefModelVal),
63       m_height(DefModelVal),
64       m_nodes_x(0),
65       m_nodes_y(0),
66       m_x_delta(0.0),
67       m_y_delta(0.0),
68       m_time_delta(0.0) {}
69
70 Model(double width, double height, double delta_n, double time_delta);
71
72 /**
73  * Set three points of hole geometry
74  * @param p1 first point
75  * @param p2 second point
76  * @param p3 third point
77  */
78 void SetHoleGeometry(Point p1, Point p2, Point p3);
79
80 /**
81  * Sets the initial conditions of the model
82  * @param init_conditions Desired initial condition
83  */
84 void SetInitialCondition(ModelNodeType init_conditions);
85
86 /*
87  * Just the few setters for restrictions, that have not specific behavior.
88  */
89 void SetOuterRestrictions(
90     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_up,
91     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_down,
92     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_left,
93     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_right)
94 ;
95 void SetOuterRestrictions(
96     const restr::BoundaryRestrictionsStorageType<ModelNodeType>
97     &restrictions);
98 void SetInnerRestrictions(
99     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restriction)
100 ;
101
102 void TimeIntegrate(double total_time,
103     solution::SolutionStorageBase<ModelNodeType> &storage,
104     ModelNodeType tube_flow);

```



```

103 void SaveResult(solution::SolutionStorageBase<ModelNodeType> &storage)
104     const {
105     storage.CommitLayer(m_mesh_ptr_present);
106 }
107 private:
108     MatrixPointerType m_mesh_ptr_present;
109     MatrixPointerType m_mesh_ptr_last;
110
111     double m_width;
112     double m_height;
113     size_t m_nodes_x;
114     size_t m_nodes_y;
115     double m_x_delta;
116     double m_y_delta;
117
118     double m_time_delta;
119
120     HoleGeometry m_hole_geometry;
121     restr::BoundaryRestrictionsStorageType<ModelNodeType> m_outer_restrictions
122     ;
123     restr::BoundaryRestrincionPointerType<ModelNodeType> m_inner_restriction;
124
125     // Bellow functions helps to determine hole and boundary points related to
126     // hole
127     [[nodiscard]] bool PointInHole(Point point) const;
128     [[nodiscard]] bool PointOnBorder(Point point) const;
129     // Calculates auxiliary values for PointInHole and PointOnBorder function
130     [[nodiscard]] std::tuple<ModelNodeType, ModelNodeType, ModelNodeType>
131     CalcCheckValues(
132         Point point) const; // sorry for that, it's just a formatter :)))
133     ModelNodeType GetInnerNeighbor(size_t x_shift, size_t y_shift);
134
135     /*
136     * Calculation methods. Just use for improve code readability and
137     * decompose layer calculation.
138     */
139     void ComputeBoundaries();
140     void ComputePlate(ModelNodeType tube_flow);
141 };
142 namespace exceptions {
143 class ModelBaseException : std::exception {
144     [[nodiscard]] const char *what() const noexcept override {
145         return "Model exception occur";
146     }
147 };
148
149 class WrongDeltaRel : std::exception {
150     [[nodiscard]] const char *what() const noexcept override {
151         return "Error: (dt / dx) ^ 2 > 1 / 2";
152     }
153 };
154 } // namespace exceptions
155 } // namespace fdm
156
157 #endif // FINITEDIFFERENCEMETHOD_MODEL_HPP_

```

Листинг 4: Model.hpp

```

1  #include "Model.hpp"
2
3  #include <iostream>
4  #include <tuple>
5  #include <vector>
6
7  #include "CalculationUtils.hpp"
8
9  namespace fdm {
10 namespace {
11 bool IsInHole(const std::vector<Model::ModelNodeType> &values) {
12     size_t sign_counter = 0;
13     for (const Model::ModelNodeType &item : values) {
14         if (item < 0) {
15             ++sign_counter;
16         }
17     }
18
19     return (sign_counter == 3 || sign_counter == 0);
20 }
21 } // anonymous namespace
22
23 Model::Model(double width, double height, double delta_n, double time_delta)
24     : m_mesh_ptr_present(MatrixBuilder().Build<ModelNodeType>()),
25       m_mesh_ptr_last(MatrixBuilder().Build<ModelNodeType>()),
26       m_width(width),
27       m_height(height),
28       m_x_delta(delta_n),
29       m_y_delta(delta_n),
30       m_time_delta(time_delta) {
31     if ((m_time_delta / m_x_delta) * (m_time_delta / m_x_delta) > 0.5) {
32         throw exceptions::WrongDeltaRel();
33     }
34     m_nodes_x = static_cast<size_t>(m_width / m_x_delta);
35     m_nodes_y = static_cast<size_t>(m_height / m_y_delta);
36     m_mesh_ptr_present->SetSize(m_nodes_y, m_nodes_x);
37     m_mesh_ptr_last->SetSize(m_nodes_y, m_nodes_x);
38
39     m_hole_geometry[0] = Point();
40     m_hole_geometry[1] = Point();
41     m_hole_geometry[2] = Point();
42 }
43
44 void Model::SetHoleGeometry(Point p1, Point p2, Point p3) {
45     m_hole_geometry[0] = p1;
46     m_hole_geometry[1] = p2;
47     m_hole_geometry[2] = p3;
48 }
49
50 void Model::SetInitialCondition(ModelNodeType init_conditions) {
51     m_mesh_ptr_present->FillMatrix(init_conditions);
52     m_mesh_ptr_last->FillMatrix(init_conditions);
53 }
54
55 void Model::SetOuterRestrictions(
56     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_up,
57     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_down,
58     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_left,
59     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_right)

```

```

    {
60     m_outer_restrictions[restr::UP_RESTRICTION] = restr_up;
61     m_outer_restrictions[restr::DOWN_RESTRICTION] = restr_down;
62     m_outer_restrictions[restr::LEFT_RESTRICTION] = restr_left;
63     m_outer_restrictions[restr::RIGHT_RESTRICTION] = restr_right;
64 }
65
66 void Model::SetOuterRestrictions(
67     const restr::BoundaryRestrictionsStorageType<ModelNodeType> &
        restrictions) {
68     m_outer_restrictions = restrictions;
69 }
70 void Model::SetInnerRestrictions(
71     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restriction)
    {
72     m_inner_restriction = restriction;
73 }
74
75 void Model::TimeIntegrate(double total_time,
76                           solution::SolutionStorageBase<ModelNodeType> &
        storage,
77                           ModelNodeType tube_flow) {
78     storage.CommitLayer(m_mesh_ptr_present);
79
80     auto time_integrate_iterations =
81         static_cast<size_t>(total_time / m_time_delta);
82
83     // Iterate time layers
84     for (size_t t = 0; t < time_integrate_iterations; ++t) {
85         m_mesh_ptr_last = m_mesh_ptr_present;
86         ComputeBoundaries();
87
88         ComputePlate(tube_flow);
89
90         storage.CommitLayer(m_mesh_ptr_present);
91     }
92     storage.CommitLayer(m_mesh_ptr_present);
93 }
94
95 void Model::ComputeBoundaries() {
96     // Traverse all boundary nodes necessary
97
98     // Firstly perform left and right boundaries
99     for (size_t i = 1; i < m_mesh_ptr_last->SizeRows() - 1; ++i) {
100         ModelNodeType T_x_left_inner = m_mesh_ptr_present->GetValue(i, 1);
101         ModelNodeType T_x_right_inner =
102             m_mesh_ptr_present->GetValue(i, m_mesh_ptr_last->SizeCols() - 2);
103
104         m_mesh_ptr_present->SetValue(
105             i, 0,
106             m_outer_restrictions[restr::LEFT_RESTRICTION]->operator()(
107                 T_x_left_inner, m_y_delta));
108         m_mesh_ptr_present->SetValue(
109             i, m_mesh_ptr_last->SizeCols() - 1,
110             m_outer_restrictions[restr::RIGHT_RESTRICTION]->operator()(
111                 T_x_right_inner, m_y_delta));
112     }
113
114     // Finally, perform up and down boundaries

```

```

115 for (size_t i = 0; i < m_mesh_ptr_last->SizeCols(); ++i) {
116     ModelNodeType T_x_down_inner = m_mesh_ptr_present->GetValue(1, i);
117     ModelNodeType T_x_up_inner =
118         m_mesh_ptr_present->GetValue(m_mesh_ptr_last->SizeRows() - 1, i);
119     m_mesh_ptr_present->SetValue(
120         0, i,
121         m_outer_restrictions[restr::DOWN_RESTRICTION]->operator()(
122             T_x_down_inner, m_x_delta));
123     m_mesh_ptr_present->SetValue(
124         m_mesh_ptr_last->SizeRows() - 1, i,
125         m_outer_restrictions[restr::UP_RESTRICTION]->operator()(T_x_up_inner
126             ,
127             m_x_delta));
128 }
129 // In fact there is no necessary dependencies for mesh boundary traversal.
130 // So you can change this order if you need.
131 }
132
133 void Model::ComputePlate(ModelNodeType tube_flow) {
134     for (size_t j = 1; j < m_mesh_ptr_last->SizeRows() - 1; ++j) {
135         for (size_t i = 1; i < m_mesh_ptr_last->SizeCols() - 1; ++i) {
136             Point curr_point(static_cast<double>(i) * m_x_delta,
137                             static_cast<double>(j) * m_y_delta);
138             if (!PointInHole(curr_point)) {
139                 if (PointOnBorder(curr_point)) {
140                     ModelNodeType inner_value = GetInnerNeighbor(i, j);
141                     m_mesh_ptr_present->SetValue(
142                         j, i, m_inner_restriction->operator()(inner_value, m_x_delta))
143                     ;
144                 } else {
145                     equations::HeatConductionParamsType<ModelNodeType>
146                         equation_parameters{m_mesh_ptr_last->GetValue(j, i),
147                                             m_mesh_ptr_last->GetValue(j, i - 1),
148                                             m_mesh_ptr_last->GetValue(j, i + 1),
149                                             m_mesh_ptr_last->GetValue(j - 1, i),
150                                             m_mesh_ptr_last->GetValue(j + 1, i),
151                                             m_time_delta,
152                                             m_x_delta,
153                                             m_y_delta,
154                                             0.1};
155                     m_mesh_ptr_present->SetValue(
156                         j, i, equations::HeatConductionProblem(equation_parameters));
157                 }
158             } else {
159                 m_mesh_ptr_present->SetValue(j, i,
160                     std::forward<ModelNodeType>(tube_flow))
161             ;
162         }
163     }
164 }
165
166 std::tuple<Model::ModelNodeType, Model::ModelNodeType, Model::ModelNodeType>
167 Model::CalcCheckValues(Point point) const {
168     ModelNodeType check_val1 = (m_hole_geometry[0].x - point.x) *
169         (m_hole_geometry[1].y - m_hole_geometry[0].
170         y) -
171         (m_hole_geometry[1].x - m_hole_geometry[0].x) *

```

```

170         (m_hole_geometry[0].y - point.y);
171 ModelNodeType check_val2 = (m_hole_geometry[1].x - point.x) *
172         (m_hole_geometry[2].y - m_hole_geometry[1].
173         y) -
174         (m_hole_geometry[2].x - m_hole_geometry[1].x) *
175         (m_hole_geometry[1].y - point.y);
176 ModelNodeType check_val3 = (m_hole_geometry[2].x - point.x) *
177         (m_hole_geometry[0].y - m_hole_geometry[2].
178         y) -
179         (m_hole_geometry[0].x - m_hole_geometry[2].x) *
180         (m_hole_geometry[2].y - point.y);
181 return {check_val1, check_val2, check_val3};
182 }
183
184 Model::ModelNodeType Model::GetInnerNeighbor(size_t x_shift, size_t y_shift)
185 {
186     auto cast_x_shift = static_cast<double>(x_shift);
187     auto cast_y_shift = static_cast<double>(y_shift);
188     Point up_neighbor((cast_x_shift + 1) * m_x_delta, cast_y_shift * m_y_delta
189     );
190     Point down_neighbor((cast_x_shift - 1) * m_x_delta, cast_y_shift *
191     m_y_delta);
192     Point right_neighbor(cast_x_shift * m_x_delta,
193     (cast_y_shift - 1) * m_y_delta);
194     Point left_neighbor(cast_x_shift * m_x_delta, (cast_y_shift + 1) *
195     m_y_delta);
196
197     if (!PointOnBorder(up_neighbor) && !PointInHole(up_neighbor)) {
198         return m_mesh_ptr_last->GetValue(y_shift, x_shift + 1);
199     }
200
201     if (!PointOnBorder(down_neighbor) && !PointInHole(down_neighbor)) {
202         return m_mesh_ptr_last->GetValue(y_shift, x_shift - 1);
203     }
204
205     if (!PointOnBorder(right_neighbor) && !PointInHole(right_neighbor)) {
206         return m_mesh_ptr_last->GetValue(y_shift + 1, x_shift);
207     }
208
209     if (!PointOnBorder(left_neighbor) && !PointInHole(left_neighbor)) {
210         return m_mesh_ptr_last->GetValue(y_shift - 1, x_shift);
211     }
212
213     return 0.0;
214 }
215
216 bool Model::PointInHole(Point point) const {
217     /*
218     * Mathematical part - vector and pseudoscalar product.
219     * Implementation - products are considered (1,2,3 - triangle vertices, 0
220     * -
221     * point):
222     * (x1-x0)*(y2-y1)-(x2-x1)*(y1-y0)
223     * (x2-x0)*(y3-y2)-(x3-x2)*(y2-y0)
224     * (x3-x0)*(y1-y3)-(x1-x3)*(y3-y0)
225     * If they are of the same sign, then the point is inside
226     * the triangle, otherwise the point is outside the triangle.
227     */
228     auto [check_val1, check_val2, check_val3] = CalcCheckValues(point);

```

```

222
223 // The comparison described above takes place in the function IsInHole
224 return IsInHole({check_val1, check_val2, check_val3});
225 }
226
227 bool Model::PointOnBorder(Point point) const {
228     /*
229      * If one of bellow values is zero, then the point
230      * lies on the side
231      */
232
233     // Check neighbor points
234     if (!PointInHole(point)) {
235         if (PointInHole(Point(point.x, point.y - m_y_delta))) {
236             return true;
237         }
238         if (PointInHole(Point(point.x, point.y + m_y_delta))) {
239             return true;
240         }
241         if (PointInHole(Point(point.x - m_x_delta, point.y))) {
242             return true;
243         }
244         if (PointInHole(Point(point.x + m_x_delta, point.y))) {
245             return true;
246         }
247
248         if (PointInHole(Point(point.x + m_x_delta, point.y - m_y_delta))) {
249             return true;
250         }
251         if (PointInHole(Point(point.x + m_x_delta, point.y + m_y_delta))) {
252             return true;
253         }
254         if (PointInHole(Point(point.x - m_x_delta, point.y - m_y_delta))) {
255             return true;
256         }
257         if (PointInHole(Point(point.x - m_x_delta, point.y + m_y_delta))) {
258             return true;
259         }
260     }
261
262     // The comparison described above takes place in the function IsInHole
263     return false;
264 }
265
266 } // namespace fdm

```

Листинг 5: Model.cpp

## 5.5 Решение задачи при помощи реализованных библиотек

```

1 #include <iomanip>
2 #include <iostream>
3 #include <memory>
4
5 #include "Model.hpp"
6
7 constexpr double n_delta = 0.45;
8 constexpr double t_delta = 0.1;
9 constexpr double scale = 1;

```

```

10 constexpr double final_n_delta = n_delta * scale;
11 constexpr double final_t_delta = t_delta * scale;
12
13 int main() {
14     std::cout << "Computing started with values:" << std::endl;
15     std::cout << "dx = dy = " << final_n_delta << std::endl;
16     std::cout << "dt = " << final_t_delta << std::endl;
17
18     fdm::Model model(6.0, 4.0, final_n_delta, final_t_delta);
19     model.SetInitialCondition(20);
20
21     fdm::restr::BoundaryRestrictionsStorageType<fdm::Model::ModelNodeType>
22         restrictions;
23     restrictions[fdm::restr::UP_RESTRICTION] = std::make_shared<
24         fdm::restr::FirstKindRestriction<fdm::Model::ModelNodeType>>(20.0);
25     restrictions[fdm::restr::DOWN_RESTRICTION] = std::make_shared<
26         fdm::restr::SecondKindRestriction<fdm::Model::ModelNodeType>>(40.0);
27     restrictions[fdm::restr::LEFT_RESTRICTION] = std::make_shared<
28         fdm::restr::SecondKindRestriction<fdm::Model::ModelNodeType>>(40.0);
29     restrictions[fdm::restr::RIGHT_RESTRICTION] = std::make_shared<
30         fdm::restr::SecondKindRestriction<fdm::Model::ModelNodeType>>(40.0);
31     fdm::restr::BoundaryRestrincionPointerType<fdm::Model::ModelNodeType>
32         inner_restr = std::make_shared<
33         fdm::restr::ThirdKindRestriction<fdm::Model::ModelNodeType>>();
34
35     model.SetOuterRestrictions(restrictions);
36     model.SetInnerRestrictions(inner_restr);
37     model.SetHoleGeometry(fdm::Model::Point(2.0, 1.0),
38         fdm::Model::Point(5.0, 1.0),
39         fdm::Model::Point(5.0, 3.0));
40
41     fdm::solution::PlaceholderStorage<fdm::Model::ModelNodeType>
42         stream_storage_placeholder;
43     fdm::solution::StaticGnuplotHeatmapStorage<fdm::Model::ModelNodeType>
44         stream_storage_gnuplot("heatmap");
45
46     model.TimeIntegrate(25.0, stream_storage_placeholder, 0);
47     model.SaveResult(stream_storage_gnuplot);
48     return 0;
49 }

```

Листинг 6: solution.cpp