



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 лабораторной 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

1	Задание	3
2	Теоретическая часть	3
2.1	Нестационарное уравнение теплопроводности	3
2.2	Метод конечных разностей	4
3	Описание работы программы	5
4	Решение задачи	6
4.1	Решение задачи с помощью разработанной программы	6
4.2	Решение задачи с помощью пакета ANSYS	6
4.3	Сравнение результатов работы программы	7
5	Код программы	7
5.1	Matrxix	7
5.2	CalculationUtils	11
5.3	SolutionStorage	13
5.4	Model	16
5.5	Решение задачи при помощи реализованных библиотек	24

1 Задание

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

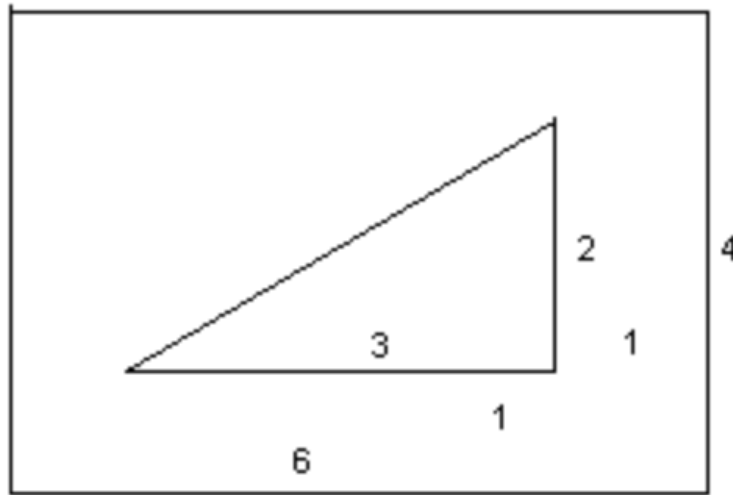


Рис. 1: Форма трубы

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 λ - 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.

Уравнения граничных условий также записываются в виде своих разностных аналогов.

ГУ 1-го рода для верхней грани трубы:

$$T = 20$$

ГУ 2-го рода для правой ли левой грани трубы:

$$\frac{T_{i-1,j} - T_{i,j}}{\Delta x} = 40,$$

$$\frac{T_{i+1,j} - T_{i,j}}{\Delta x} = 40$$

ГУ 2-го рода для нижней грани трубы:

$$\frac{T_{i,j-1} - T_{i,j}}{\Delta y} = 40$$

На границах отверстия, параллельных осям системы координат:

$$\frac{T_{i,j-1} - T_{i,j}}{\Delta y} = T_{i,j}$$

$$\frac{T_{i+1,j} - T_{i,j}}{\Delta x} = T_{i,j}$$

Для записи ГУ на наклонной грани отверстия, необходимо учесть, что поток должен выходить из тела по нормали. Для наглядности, на рисунке 2 изображена сетка в масштабе. На нем видно, как узлы располагаются на границах отверстия.

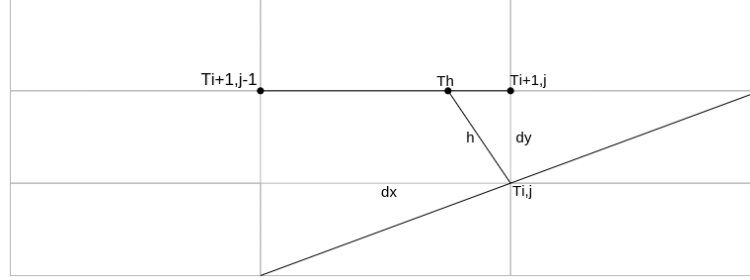


Рис. 2: Графическое изображение результатов расчетов с шагом 0.1

Введем величину $h = \frac{\Delta y}{\sin \phi}$, где ϕ - угол, между Δy и h . Очевидно, что h , в общем случае, не проходит через узлы сетки. В связи с этим найдем значение в точке T_h , которая находится на пересечении h и прямой между $T_{i+1,j-1}$ и $T_{i+1,j}$. Определим это значение при помощи уравнения прямой, фактически проведем интерполяцию, то есть:

$$T_h = (1 - \mu)T_{i+1,j-1} - \mu T_{i+1,j},$$

где $\mu = h \cos \phi$. Таким образом ГУ 3-го рода на наклонной грани можно записать так:

$$\frac{T_h - T_{i,j}}{h} = T_{i,j}$$

3 Описание работы программы

Решение нестационарного уравнения теплопроводности сводится к тому, чтобы на всех временных слоях получить значения во всех узлах сетки заданой 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 большого двумерного массива 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.

Для правильного решения поставленной задачи, при вычислении значения в узле, программа должна понимать, накладывается ли на рассчитываемый узел ГУ. И, если для границ снаружи трубы все достаточно тривиально, то, чтобы рассчитать значения на границах отверстия внутри трубы, необходимо воспользоваться алгоритмом определения таких граничных узлов. Как ранее упоминалось, шаг приращения по пространственным

координатам выбирается таким образом, чтобы узлы расчетной сетки оказывались на границах отверстия. Зная размеры внутреннего отверстия, мы легко можем определить координаты его угловых точек. Теперь, вычислив значения следующих выражений:

$$v_1 = (x_1 - x_0) * (y_2 - y_1) - (x_2 - x_1) * (y_1 - y_0),$$

$$v_2 = (x_2 - x_0) * (y_3 - y_2) - (x_3 - x_2) * (y_2 - y_0),$$

$$v_3 = (x_3 - x_0) * (y_1 - y_3) - (x_1 - x_3) * (y_3 - y_0),$$

где (x_0, y_0) - координаты интересующей нас точки, $(x_i, y_i), i = 1, 2, 3$ - координаты точек вершин треугольного отверстия. В результате мы получаем три значения v_1, v_2, v_3 . Если одно из этих значения равно нулю, то точка лежит на границе отверстия. Более того, если все три значения имеют один знак, то точка лежит внутри треугольного отверстия.

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 4.

161.426	149.426	137.563	125.643	113.487	100.834	87.3653	72.8692	58.0662	48.5994	43.3433	40.8122	40.1321	41.0435	43.8561	49.495	59.4755	74.8162	89.1116	101.112
149.426	137.529	125.663	113.736	101.571	88.9064	75.424	60.9122	46.0927	36.6149	31.3522	28.8174	28.1351	29.0453	31.8574	37.496	47.4765	62.8174	77.1128	89.1116
138.616	126.716	114.899	103.072	91.0574	78.5702	65.1332	49.7111	29.0476	20.5721	16.7231	15.0215	14.5757	15.1638	17.0448	21.1656	30.1279	51.8765	67.4235	79.4223
128.815	116.911	105.192	93.5738	81.8981	69.9417	57.4364	44.1819	0	0	0	0	0	0	0	0	0	47.1481	61.2935	73.2925
119.873	107.964	96.3904	85.0723	73.8667	62.5934	51.0874	39.298	0	0	0	0	0	0	0	0	0	44.0912	57.3194	69.3185
111.575	99.6587	88.2756	77.3397	66.6993	56.1675	45.9881	35.0677	13.0702	0	0	0	0	0	0	0	0	41.9858	54.5822	66.5815
103.64	91.7143	80.5767	70.1195	60.1536	50.422	40.5491	29.7349	15.3083	11.7695	0	0	0	0	0	0	0	40.3449	52.449	64.4484
95.7136	83.7777	72.9657	63.1273	54.0244	45.3815	36.9105	28.3541	19.8751	14.8563	11.4279	0	0	0	0	0	0	38.7888	50.4259	62.4254
87.3476	75.4083	65.0407	56.0179	47.9954	40.6565	33.7586	27.2017	21.2007	16.5072	12.3969	7.31789	5.62915	0	0	0	0	36.9573	48.0449	60.0445
77.9532	65.9935	56.3182	48.4143	41.7426	35.8926	30.6015	25.7583	21.4165	17.7187	14.4324	11.2953	9.12202	7.01694	0	0	0	34.427	44.7554	56.7551
66.6987	54.726	46.234	39.963	35.0186	30.8779	27.2564	24.0231	21.1497	18.6382	16.4023	14.3624	12.5844	10.6238	7.66041	5.89263	0	30.613	39.7971	51.7969
52.2457	40.2595	34.2011	30.4415	27.7246	25.5504	23.6983	22.0712	20.6332	19.3666	18.2369	17.2089	16.2576	15.2513	14.1329	13.6245	14.4754	24.2781	32.0247	44.0246
20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20

Рис. 3: Результат работы программы в табличном виде с шагом $h = 0.5$

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 5.

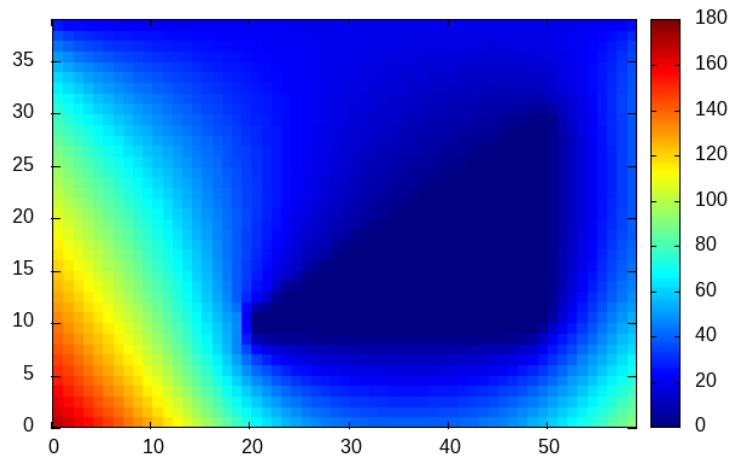


Рис. 4: Графическое изображение результатов расчетов с шагом 0.1

4.3 Сравнение результатов работы программы

На основе полученных распределений температур, представленных на рисунке 5 и рисунке 3, 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 };

```

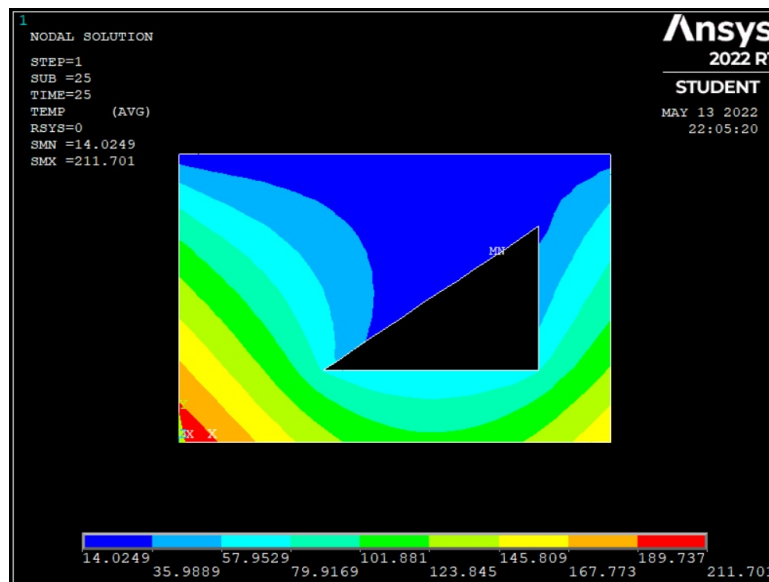


Рис. 5: Графическое изображение результатов расчетов с помощью ANSYS

```

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) ->
81         size_t {
82             return row * Cols + col;
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          ) {}
29      ModelNodeType operator()(ModelNodeType inner, double delta) override {
30          return m_constant;
31      }
32  private:
33      double m_constant;
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  private:
45      double m_constant;
46  };
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
93  * @param params pass tuple of Ti,j, Ti-1,j, Ti+1,j, Ti,j-1, Ti,j+1,
94  * delta
95  * t, delta x, delta y and a
96  * @return predict node Ti,j value on new time layer
97  */
98 template <typename ModelNodeType>
99 ModelNodeType HeatConductionProblem(
100     const HeatConductionParamsType<ModelNodeType> &params) {
101     auto [T_curr, T_x_past, T_x_next, T_y_past, T_y_next, dt, dx, dy, a] =
102         params;
103     double dx2 = dx * dx;
104     double dy2 = dy * dy;
105     ModelNodeType Dx = (T_x_next - 2 * T_curr + T_x_past) / dx2;
106     ModelNodeType Dy = (T_y_next - 2 * T_curr + T_y_past) / dy2;
107     ModelNodeType D = Dx + Dy;
108     return a * dt * D + T_curr;
109 }
110 } // namespace equations
111 } // namespace fdm
112 #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     }
24 }

```

```

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
30 } // namespace exceptions
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(

```

```

78     const StringType &file_prefix)
79     : m_file_output_prefix(file_prefix) {
80     m_plot_file_name =
81         std::string(m_file_output_prefix) + std::string(plot_extension);
82     m_config_file_name =
83         std::string(m_file_output_prefix) + std::string(config_extension);
84
85     m_plot_output.open(m_plot_file_name.data());
86     m_config_output.open(m_config_file_name.data());
87     if (!m_plot_output.is_open() || !m_config_output.is_open()) {
88         throw exceptions::FileNotOpenException();
89     }
90     WriteConfig();
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 private:
112     StringType m_file_output_prefix;
113     StringType m_plot_file_name;
114     StringType m_config_file_name;
115     std::ofstream m_config_output;
116     std::ofstream m_plot_output;
117
118     StringType plot_extension = ".plt";
119     StringType config_extension = ".cfg";
120     StringType mapping_extension = ".png";
121     StringType default_file_prefix = "def_output";
122
123     void WriteConfig() {
124         StringType terminal_def{"set terminal png\n"};
125         StringType output_file_def{
126             "set output '" + std::string(m_file_output_prefix) +
127             std::string(mapping_extension) + "'\n"};
128         StringType scale_def{
129             "set autoscale yfix\nset autoscale xfix\n"};
130         StringType palette_def{
131             "set palette defined (0 0 0 0.5, 1 0 0 1, 2 0 0.5 1, 3 0 1 1, 4 0.5
132             1 "
133             "0.5, 5 1 1 0, 6 1 0.5 0, 7 1 0 0, 8 0.5 0 0)\n"};
134         StringType map_type_def{"set pm3d map\n"};
135         StringType splot_def{

```

```

135     "splot '" + std::string(m_plot_file_name) + "' matrix notitle\n");
136     m_config_output << terminal_def;
137     m_config_output << output_file_def.data();
138     m_config_output << scale_def;
139     m_config_output << palette_def;
140     m_config_output << map_type_def;
141     m_config_output << splot_def.data();
142 }
143 };
144
145 template <typename MeshNodesType>
146 class PlaceholderStorage : public SolutionStorageBase<MeshNodesType> {
147 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 void TimeIntegrate(double total_time,
102     solution::SolutionStorageBase<ModelNodeType> &storage,
103     ModelNodeType tube_flow);
104 void SaveResult(solution::SolutionStorageBase<ModelNodeType> &storage)
105     const {
106     storage.CommitLayer(m_mesh_ptr_present);
107 }
108 private:
109     MatrixPointerType m_mesh_ptr_present;
110     MatrixPointerType m_mesh_ptr_last;
111
112     double m_width;
113     double m_height;
114     size_t m_nodes_x;
115     size_t m_nodes_y;
116     double m_x_delta;
117     double m_y_delta;
118
119     double m_time_delta;
120
121     HoleGeometry m_hole_geometry;
122     restr::BoundaryRestrictionsStorageType<ModelNodeType> m_outer_restrictions
123     ;
124     restr::BoundaryRestrincionPointerType<ModelNodeType> m_inner_restriction;
125
126     // Bellow functions helps to determine hole and boundary points related to
127     // hole
128     [[nodiscard]] bool PointInHole(Point point) const;
129     [[nodiscard]] bool PointOnBorder(Point point) const;
130     // Calculates auxiliary values for PointInHole and PointOnBorder function
131     [[nodiscard]] std::tuple<ModelNodeType, ModelNodeType, ModelNodeType>
132     CalcCheckValues(
133         Point point) const; // sorry for that, it's just a formatter :)))
134     ModelNodeType GetInnerNeighbor(size_t x_shift, size_t y_shift);
135
136     /*
137     * Calculation methods. Just use for improve code readability and
138     * decompose layer calculation.
139     */
140     void ComputeBoundaries();
141     void ComputePlate(ModelNodeType tube_flow);
142 };
143
144 namespace exceptions {
145     class ModelBaseException : std::exception {
146     [[nodiscard]] const char *what() const noexcept override {
147         return "Model exception occur";
148     }
149 }

```

```

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 {
61     m_outer_restrictions[restr::UP_RESTRICTION] = restr_up;
62     m_outer_restrictions[restr::DOWN_RESTRICTION] = restr_down;
63     m_outer_restrictions[restr::LEFT_RESTRICTION] = restr_left;
64     m_outer_restrictions[restr::RIGHT_RESTRICTION] = restr_right;
65 }
66
67 void Model::SetOuterRestrictions(
68     const restr::BoundaryRestrictionsStorageType<ModelNodeType> &
69     restrictions) {
70     m_outer_restrictions = restrictions;
71 }
72
73 void Model::SetInnerRestrictions(
74     const restr::BoundaryRestrincionPointerType<ModelNodeType> &restriction)
75 {
76     m_inner_restriction = restriction;
77 }
78
79 void Model::TimeIntegrate(double total_time,
80     solution::SolutionStorageBase<ModelNodeType> &
81     storage,
82     ModelNodeType tube_flow) {
83     storage.CommitLayer(m_mesh_ptr_present);
84
85     auto time_integrate_iterations =
86         static_cast<size_t>(total_time / m_time_delta);
87
88     // Iterate time layers
89     for (size_t t = 0; t < time_integrate_iterations; ++t) {
90         m_mesh_ptr_last = m_mesh_ptr_present;
91         ComputeBoundaries();
92
93         ComputePlate(tube_flow);
94
95         storage.CommitLayer(m_mesh_ptr_present);
96     }
97     storage.CommitLayer(m_mesh_ptr_present);
98 }
99
100 void Model::ComputeBoundaries() {
101     // Traverse all boundary nodes necessary
102
103     // Firstly perform left and right boundaries
104     for (size_t i = 1; i < m_mesh_ptr_last->SizeRows() - 1; ++i) {
105         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
130 // In fact there is no necessary dependencies for mesh boundary traversal.
131 // So you can change this order if you need.
132 }
133
134 void Model::ComputePlate(ModelNodeType tube_flow) {
135     for (size_t j = 1; j < m_mesh_ptr_last->SizeRows() - 1; ++j) {
136         for (size_t i = 1; i < m_mesh_ptr_last->SizeCols() - 1; ++i) {
137             Point curr_point(static_cast<double>(i) * m_x_delta,
138                             static_cast<double>(j) * m_y_delta);
139             if (!PointInHole(curr_point)) {
140                 if (PointOnBorder(curr_point)) {
141                     ModelNodeType inner_value1 = GetInnerNeighbor(i + 1, j - 1);
142                     ModelNodeType inner_value2 = GetInnerNeighbor(i + 1, j);
143                     ModelNodeType interpolate_inner_value = (inner_value1 + inner_value2)
144 / 2;
145                     m_mesh_ptr_present->SetValue(
146                         j, i, m_inner_restriction->operator()(interpolate_inner_value,
147 m_x_delta));
148                 } else {
149                     equations::HeatConductionParamsType<ModelNodeType>
150                     equation_parameters{m_mesh_ptr_last->GetValue(j, i),
151                                         m_mesh_ptr_last->GetValue(j, i - 1),
152                                         m_mesh_ptr_last->GetValue(j, i + 1),
153                                         m_mesh_ptr_last->GetValue(j - 1, i),
154                                         m_mesh_ptr_last->GetValue(j + 1, i),
155                                         m_time_delta,
156                                         m_x_delta,
157                                         m_y_delta,
158                                         0.1};
159                     m_mesh_ptr_present->SetValue(

```

```

157         j, i, equations::HeatConductionProblem(equation_parameters));
158     }
159     } else {
160         m_mesh_ptr_present->SetValue(j, i,
161                                     std::forward<ModelNodeType>(tube_flow))
162     ;
163     }
164 }
165 }
166
167 std::tuple<Model::ModelNodeType, Model::ModelNodeType, Model::ModelNodeType>
168 Model::CalcCheckValues(Point point) const {
169     ModelNodeType check_val1 = (m_hole_geometry[0].x - point.x) *
170                               (m_hole_geometry[1].y - m_hole_geometry[0].
171                                y) -
172                               (m_hole_geometry[1].x - m_hole_geometry[0].x) *
173                               (m_hole_geometry[0].y - point.y);
174     ModelNodeType check_val2 = (m_hole_geometry[1].x - point.x) *
175                               (m_hole_geometry[2].y - m_hole_geometry[1].
176                                y) -
177                               (m_hole_geometry[2].x - m_hole_geometry[1].x) *
178                               (m_hole_geometry[1].y - point.y);
179     ModelNodeType check_val3 = (m_hole_geometry[2].x - point.x) *
180                               (m_hole_geometry[0].y - m_hole_geometry[2].
181                                y) -
182                               (m_hole_geometry[0].x - m_hole_geometry[2].x) *
183                               (m_hole_geometry[2].y - point.y);
184     return {check_val1, check_val2, check_val3};
185 }
186
187 Model::ModelNodeType Model::GetInnerNeighbor(size_t x_shift, size_t y_shift)
188 {
189     auto cast_x_shift = static_cast<double>(x_shift);
190     auto cast_y_shift = static_cast<double>(y_shift);
191     Point up_neighbor((cast_x_shift + 1) * m_x_delta, cast_y_shift * m_y_delta
192 );
193     Point down_neighbor((cast_x_shift - 1) * m_x_delta, cast_y_shift *
194 m_y_delta);
195     Point right_neighbor(cast_x_shift * m_x_delta,
196 (cast_y_shift - 1) * m_y_delta);
197     Point left_neighbor(cast_x_shift * m_x_delta, (cast_y_shift + 1) *
198 m_y_delta);
199
200     if (!PointOnBorder(up_neighbor) && !PointInHole(up_neighbor)) {
201         return m_mesh_ptr_last->GetValue(y_shift, x_shift + 1);
202     }
203
204     if (!PointOnBorder(down_neighbor) && !PointInHole(down_neighbor)) {
205         return m_mesh_ptr_last->GetValue(y_shift, x_shift - 1);
206     }
207
208     if (!PointOnBorder(right_neighbor) && !PointInHole(right_neighbor)) {
209         return m_mesh_ptr_last->GetValue(y_shift + 1, x_shift);
210     }
211
212     if (!PointOnBorder(left_neighbor) && !PointInHole(left_neighbor)) {
213         return m_mesh_ptr_last->GetValue(y_shift - 1, x_shift);
214     }
215 }

```

```

208
209     return 0.0;
210 }
211
212 bool Model::PointInHole(Point point) const {
213     /*
214      * Mathematical part - vector and pseudoscalar product.
215      * Implementation - products are considered (1,2,3 - triangle vertices, 0
216      * -
217      * point):
218      * (x1-x0)*(y2-y1)-(x2-x1)*(y1-y0)
219      * (x2-x0)*(y3-y2)-(x3-x2)*(y2-y0)
220      * (x3-x0)*(y1-y3)-(x1-x3)*(y3-y0)
221      * If they are of the same sign, then the point is inside
222      * the triangle, otherwise the point is outside the triangle.
223      */
224     auto [check_val1, check_val2, check_val3] = CalcCheckValues(point);
225
226     // The comparison described above takes place in the function IsInHole
227     return IsInHole({check_val1, check_val2, check_val3});
228 }
229
230 bool Model::PointOnBorder(Point point) const {
231     /*
232      * If one of bellow values is zero, then the point
233      * lies on the side
234      */
235
236     // Check neighbor points
237     if (!PointInHole(point)) {
238         if (PointInHole(Point(point.x, point.y - m_y_delta))) {
239             return true;
240         }
241         if (PointInHole(Point(point.x, point.y + m_y_delta))) {
242             return true;
243         }
244         if (PointInHole(Point(point.x - m_x_delta, point.y))) {
245             return true;
246         }
247         if (PointInHole(Point(point.x + m_x_delta, point.y))) {
248             return true;
249         }
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         if (PointInHole(Point(point.x - m_x_delta, point.y + m_y_delta))) {
261             return true;
262         }
263     }
264
265     // The comparison described above takes place in the function IsInHole
266     return false;

```

```

266 }
267
268 } // 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 = 0.3;
8 constexpr double t_delta = 0.07;
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