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 _____ Дунайцев А. И
подпись, дата

Преподаватель _____ Трудоношин В. А.
подпись, дата

Москва, 2022

Содержание

1 Задание

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

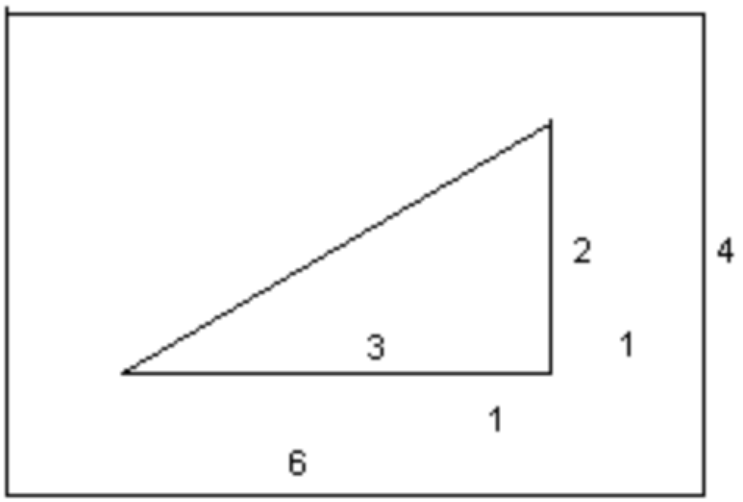

Рис. 1: Форма трубы

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 урпанение имеет следующий вид:

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

где λ - коэфициент 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 аналагом, который представляет сосбой сетку, в узлах которой находятся исследуемые значения. Каждый узел соответствует значению в определенной простарнственной координате исходного объекта, таким образом координата узла вычисляется следующим $(x_i, y_j) = (i\Delta x, j\Delta y)$, где $i \in [0, ..., n]$, $j \in [0, ...m]$, а n, m - количество узлов вдоль оси абсцисс и оси ординат соответственно.

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

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

Вторая производная

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

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, соответствуюший этой переменной.

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

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

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 коодринат:

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

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

Для записи ГУ на наклонной грани отверстия введем переменную $h = \sqrt{\Delta x^2 + \Delta y^2}$. Тогда ГУ 3-го рода на этой грани можно записать так:

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

3 Описание работы программы

Решение нестационарного уравнения теплопроводности сводится к тому, чтобы на всех временных слоях получить 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 рассчетной сетки оказывались на границах отверстия. Зная размеры внутреннего отверстия, мы легко можем определить координаты его угловых точек. Теперь, вычислив значения следующих выражений:

$$v_1 = (x_1 - x0) * (y_2 - y_1) - (x_2 - x_1) * (y_1 - y_0),$$
$$v_2 = (x_2 - x_0) * (y_3 - y_2) - (x_3 - x_2) * (y_2 - y_0),$$
$$v_3 = (x_3 - x_0) * (y_1 - y_3) - (x_1 - x_3) * (y_3 - y_0),$$

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 сводится только к рассчетам на всех временных слоях, а не попыткам сохранить их в оперативной памяти компьютера, поэтому реализованый класс модели не отвечает за хранение вычислений. Он лишь принимает полиморфный объект хранилища, который отвечает за сохранение результата рассчетов, 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 виляют 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	40
126.993	107.072	84.0267	56.0178	0	0	0	0	0	0	0	20
112.009	92.0833	70.5678	47.0452	0	0	0	0	0	0	0	20
100.309	80.3746	60.531	40.354	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	20
78.9532	58.9911	43.7958	31.3868	20.9235	14.8128	9.87521	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	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

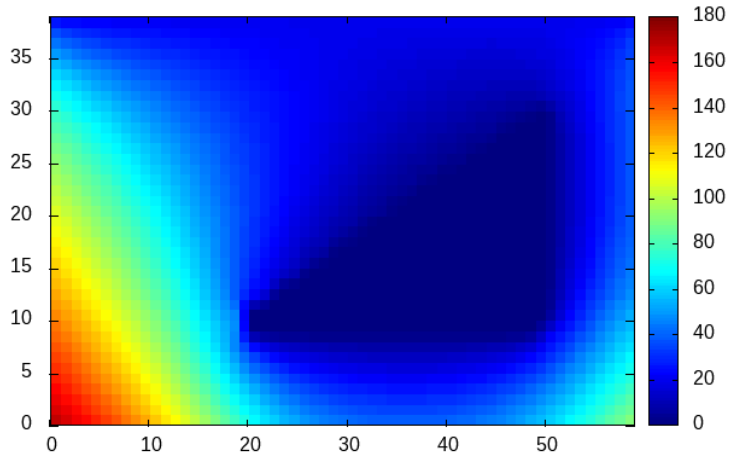

Рис. 3: Графическое изображение результатов рассчетов с шагом 0.1

4.2 Решение задачи с помощью пакета ANSYS

Для решения задачи теплопроводности трубы средствами покета 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.

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.

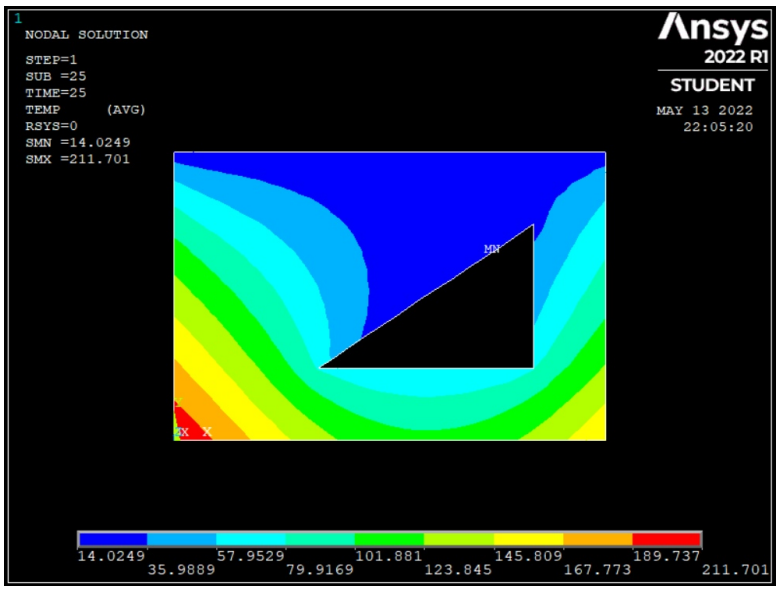

Рис. 4: Графическое изображение результатов рассчетов с помощью ANSYS

5 Код программы

5.1 Matrxix

```
#ifndef FINITEDIFFERENCEMETHOD_MATRIX_HPP_
#define FINITEDIFFERENCEMETHOD_MATRIX_HPP_

#include <algorithm>
#include <array>
#include <cstddef>
#include <exception>
#include <memory>
#include <type_traits>
#include <vector>

namespace mtrx {
namespace exceptions {
class BaseMatrixException : public std::exception {
  [[nodiscard]] const char *what() const noexcept override {
    return "Matrix error occur";
  }
};

class MatrixSizeException : public BaseMatrixException {
  [[nodiscard]] const char *what() const noexcept override {
    return "Matrix size error occur. You may have gone beyond the matrix.";
  }
};
}  // namespace exceptions

/**
 * I do not really want to complicate the code in the context of my tasks.
 * Therefore, I add the implementation of the store and accessor as much as
 * possible. But MatrixBase provide the simple interface, that using in
 * FDM implementation.
 *
```

```
 * @note All derived classes have to necessary define Type
 *
 * @tparam Tp matrix elements type
 */
namespace base {
template <typename Tp>
class MatrixBase {
 public:
  using Type = Tp;
  virtual void SetValue(size_t row, size_t col, Tp &&val) = 0;
  virtual const Tp &GetValue(size_t row, size_t col) const = 0;
  virtual size_t SizeRows() const = 0;
  virtual size_t SizeCols() const = 0;
  [[maybe_unused]] virtual void FillMatrix([[maybe_unused]] Tp val) {}

  virtual ~MatrixBase() = default;
};

template <typename Tp>
class MatrixDynamicBase : public MatrixBase<Tp> {
 public:
  using Type = Tp;
  virtual void SetRows(size_t rows) = 0;
  virtual void SetCols(size_t cols) = 0;
  void SetSize(size_t rows, size_t cols) {
    this->SetRows(rows);
    this->SetCols(cols);
  }

  virtual ~MatrixDynamicBase() = default;
};
}  // namespace base

/**
 * Simplest matrix implementation. Matrix class in this program maintains
 * only basic operations and using as structure for easy to store and
 * access values.
 *
 * @tparam Tp matrix elements type
 * @tparam Cols amount of columns
 * @tparam Rows amount of m_rows
 */
template <typename Tp, size_t Rows, size_t Cols>
class Matrix : public base::MatrixBase<Tp> {
 public:
  using Type = Tp;
  using MatrixStorageType = std::array<Type, Cols * Rows>;
  static constexpr auto MatrixAccessor = [](size_t row, size_t col) ->
   size_t {
    return row * Cols + col;
  };

  void SetValue(size_t row, size_t col, Tp &&val) override;
  const Tp &GetValue(size_t row, size_t col) const override;
  [[nodiscard]] virtual size_t SizeRows() const override { return Rows; }
  [[nodiscard]] virtual size_t SizeCols() const override { return Cols; }

  Matrix() = default;

```

```
 private:
  MatrixStorageType m_storage;

  [[nodiscard]] bool CheckAccess(size_t row, size_t col) const {
    return col < Cols && row < Rows;
  }
};

template <typename Tp, size_t Rows, size_t Cols>
void Matrix<Tp, Rows, Cols>::SetValue(size_t row, size_t col, Tp &&val) {
  if (!CheckAccess(row, col)) {
  }
  m_storage[MatrixAccessor(row, col)] = std::forward<Tp>(val);
}

template <typename Tp, size_t Rows, size_t Cols>
const Tp &Matrix<Tp, Rows, Cols>::GetValue(size_t row, size_t col) const {
  if (!CheckAccess(row, col)) {
  }
  return m_storage[MatrixAccessor(row, col)];
}

/**
 * This matrix builder is created only because I started implemented
 * current library with only static matrix
 */
template <typename MatrixClass>
class MatrixCreatorStatic {
 public:
  using BaseType = base::MatrixBase<typename MatrixClass::Type>;
  using MatrixPointer = std::unique_ptr<BaseType>;

  MatrixPointer operator()() {
    static_assert(std::is_base_of_v<BaseType, MatrixClass>);
    return std::make_unique<MatrixClass>();
  }
};

template <typename Tp>
class MatrixDynamic : public base::MatrixDynamicBase<Tp> {
 public:
  using Type = Tp;
  using MatrixStorageType = std::vector<Type>;

  void SetRows(size_t rows) override {
    m_rows = rows;
    m_storage.resize(m_rows * m_cols);
  }
  void SetCols(size_t cols) override {
    m_cols = cols;
    m_storage.resize(m_rows * m_cols);
  }
  void SetSize(size_t rows, size_t cols) {
    m_rows = rows;
    m_cols = cols;
    m_storage.resize(rows * cols);
  }
  [[nodiscard]] size_t SizeRows() const override { return m_rows; }
  [[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 };
```

```
} // namespace mtrx

#endif // FINITEDIFFERENCEMETHOD_MATRIX_HPP_
```

Листинг 1: Matrix.hpp

5.2 CalculationUtils

```
#ifndef FINITEDIFFERENCEMETHOD_CALCULATIONUTILS_HPP_
#define FINITEDIFFERENCEMETHOD_CALCULATIONUTILS_HPP_

#include <memory>
#include <tuple>

namespace fdm {
/*
 * I decided to make the restrictions separate from the model class,
 * since if desired, these classes can be reused for other specific models.
 */
namespace restr {
template <typename ModelNodeType>
class BaseRestriction {
 public:
  virtual ModelNodeType operator()(ModelNodeType, double delta) = 0;
  virtual ~BaseRestriction() = default;
};

/**
 * As we know, first kind restriction return constant value not depend
 * on boundary derivative value.
 */
template <typename ModelNodeType>
class FirstKindRestriction : public BaseRestriction<ModelNodeType> {
 public:
  explicit FirstKindRestriction(double constant = 0.0) : m_constant(constant
    ) {}
  ModelNodeType operator()(ModelNodeType inner, double delta) override {
    return m_constant;
  }

 private:
  double m_constant;
};

template <typename ModelNodeType>
class SecondKindRestriction : public BaseRestriction<ModelNodeType> {
 public:
  explicit SecondKindRestriction(double constant = 0.0)
      : m_constant(constant) {}
  ModelNodeType operator()(ModelNodeType inner, double delta) override {
    return inner + m_constant * delta;
  }

 private:
  double m_constant;
};

template <typename ModelNodeType>
class ThirdKindRestriction : public BaseRestriction<ModelNodeType> {
```

```
 public:
  ModelNodeType operator()(ModelNodeType inner, double delta) override {
    return inner / (1 + delta);
  }
};

/*
 * Restriction is the callable object, that calculate boundary values.
 * Parameters are passed to the function, in the bellow signature,
 * in order of taking the derivative from inside to outside. Last parameter
 * represent differential specific derivative increment value.
 */
template <typename ModelNodeType>
using BoundaryRestrincionType = BaseRestriction<ModelNodeType>;
template <typename ModelNodeType>
using BoundaryRestrincionPointerType =
    std::shared_ptr<BoundaryRestrincionType<ModelNodeType>>;
// Restrictions on the outer boundaries of tube.
template <typename ModelNodeType>
using BoundaryRestrictionsStorageType =
    std::array<BoundaryRestrincionPointerType<ModelNodeType>, 4>;
// Order of restrictions
constexpr size_t UP_RESTRICTION = 0;
constexpr size_t DOWN_RESTRICTION = 1;
constexpr size_t LEFT_RESTRICTION = 2;
constexpr size_t RIGHT_RESTRICTION = 3;
}  // namespace restr

namespace equations {
/**
 * @param params pass tuple of T_i_j, T_i-1_j, T_i+1,j, T_i_j-1, T_i_j+1,
   delta
 * t, delta x, delta y and a
 */
template <typename ModelNodeType>
using HeatConductionParamsType =
    std::tuple<ModelNodeType, ModelNodeType, ModelNodeType, ModelNodeType,
               ModelNodeType, double, double, double, double>;
/**
 * Solve the non-stationary problem of heat conduction by an explicit method
   .
 * @tparam ModelNodeType model node type
 * @param params pass tuple of T_i_j, T_i-1_j, T_i+1,j, T_i_j-1, T_i_j+1,
   delta
 * t, delta x, delta y and a
 * @return predict node T_i_j value on new time layer
 */
template <typename ModelNodeType>
ModelNodeType HeatConductionProblem(
    const HeatConductionParamsType<ModelNodeType> &params) {
  auto [T_curr, T_x_past, T_x_next, T_y_past, T_y_next, dt, dx, dy, a] =
   params;
  double dx2 = dx * dx;
  double dy2 = dy * dy;
  ModelNodeType Dx = (T_x_next - 2 * T_curr + T_x_past) / dx2;
  ModelNodeType Dy = (T_y_next - 2 * T_curr + T_y_past) / dy2;
  ModelNodeType D = Dx + Dy;
  return a * dt * D + T_curr;
}
```

```
} // namespace equations

} // namespace fdm

#endif // FINITEDIFFERENCEMETHOD_CALCULATIONUTILS_HPP_
```

Листинг 2: CalculationUtils.hpp

5.3 SolutionStorage

```
#ifndef FINITEDIFFERENCEMETHOD_SOLUTIONSTORAGE_HPP_
#define FINITEDIFFERENCEMETHOD_SOLUTIONSTORAGE_HPP_

#include <exception>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <memory>
#include <string_view>

#include "Matrix.hpp"

namespace fdm {
namespace solution {
namespace exceptions {
class SolutionStorageException : public std::exception {
 public:
  [[nodiscard]] const char *what() const noexcept override {
    return "Storage exception occur";
  }
};

class FileNotOpenException : public std::exception {
 public:
  [[nodiscard]] const char *what() const noexcept override {
    return "One of files (configuration or plot) was not open";
  }
};

} // namespace exceptions
/**
 * Basic model's mesh storing class. Class provides functions, that helps
 * to manage mesh optimal storage. This class is necessary to use if you
 * dont want to lose your calculations data, because Model class don't
 * store all time integrated layers.
 *
 * @tparam MeshNodesType specify mesh nodes type
 */
template <typename MeshNodesType>
class SolutionStorageBase {
 public:
  using MeshType = mtrx::base::MatrixBase<MeshNodesType>;
  using MeshPointerType = std::shared_ptr<MeshType>;

  virtual void CommitLayer(const MeshPointerType &mesh_ptr) = 0;
  virtual ~SolutionStorageBase() = default;
};

/**
```

```
 * Simplest storage, that commit all integrated layers of
 * model in standard output
 */
template <typename MeshNodesType>
class StandardStreamStorage : public SolutionStorageBase<MeshNodesType> {
 public:
  std::ostream &StandardStreamDefinition = std::cout;

  void CommitLayer(
      const typename SolutionStorageBase<MeshNodesType>::MeshPointerType
          &mesh_ptr) override {
    //    StandardStreamDefinition.precision(2);
    for (size_t i = 0; i < mesh_ptr->SizeRows(); ++i) {
      for (size_t j = 0; j < mesh_ptr->SizeCols(); ++j) {
        StandardStreamDefinition << mesh_ptr->GetValue(i, j) << ' ';
      }
      StandardStreamDefinition << '\n';
    }
    StandardStreamDefinition << '\n';
  }
};

template <typename MeshNodesType>
class StaticGnuplotHeatmapStorage : public SolutionStorageBase<MeshNodesType
    > {
 public:
  using StringType = std::string;

  explicit StaticGnuplotHeatmapStorage(
      const StringType &file_prefix)
      : m_file_output_prefix(file_prefix) {
    m_plot_file_name =
        std::string(m_file_output_prefix) + std::string(plot_extension);
    m_config_file_name =
        std::string(m_file_output_prefix) + std::string(config_extension);

    m_plot_output.open(m_plot_file_name.data());
    m_config_output.open(m_config_file_name.data());
    if (!m_plot_output.is_open() || !m_config_output.is_open()) {
      throw exceptions::FileNotOpenException();
    }
    WriteConfig();
  }

  void CommitLayer(
      const typename SolutionStorageBase<MeshNodesType>::MeshPointerType
          &mesh_ptr) override {
    std::ostream &StandardStreamDefinition = m_plot_output;
    for (size_t i = 0; i < mesh_ptr->SizeRows(); ++i) {
      for (size_t j = 0; j < mesh_ptr->SizeCols(); ++j) {
        StandardStreamDefinition << std::setw(8) << mesh_ptr->GetValue(i, j)
    << ' ';
      }
      StandardStreamDefinition << '\n';
    }
    StandardStreamDefinition << '\n';
  }

  ~StaticGnuplotHeatmapStorage() override {
```

```
107     m_plot_output.close();
108     m_config_output.close();
109   }
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
    1 "
132         "0.5, 5 1 1 0, 6 1 0.5 0, 7 1 0 0, 8 0.5 0 0)\n"};
133   StringType map_type_def{"set pm3d map\n"};
134   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>
```

```cpp
#include <exception>
#include <fstream>
#include <functional>
#include <iostream>
#include <tuple>
#include <utility>
#include <vector>

#include "CalculationUtils.hpp"
#include "Matrix.hpp"
#include "SolutionStorage.hpp"

// As you'll see, I'm a big fan of readable aliases. Don't swear if this
    makes
// my code unreadable. =)

namespace fdm {
/**
 * Since finite difference coding in general cannot be done for
 * free-form geometry, I wrote a model that solves the
 * non-stationary heat conduction problem in an explicit way on
 * the geometry of a tube with a triangular hole.
 */
class Model {
 public:
  /*
   * If you wish to change the implementation of the class with
   * your own data types that implement the interface declared
   * in the Matrix.h file, change the aliases declared here.
   * This is done so as not to complicate the understanding of
   * the program code.
   */
  using ModelNodeType = double;
  constexpr static ModelNodeType DefModelVal = 0.0;
  using MatrixBuilder = mtrx::MatrixCreatorDynamic;
  using MatrixPointerType = MatrixBuilder::Pointer<ModelNodeType>;

  /*
   * Actually I could use std::pair, but in my opinion Point
   * is more representative.
   */
  struct Point {
  ModelNodeType x;
  ModelNodeType y;
  Point() : x(DefModelVal), y(DefModelVal) {}
  Point(ModelNodeType _x, ModelNodeType _y) : x(_x), y(_y) {}
  };

  /*
   * Triangular hole in the center of tube. I think
   * good idea to store it in fixed size array.
   */
  using HoleGeometry = std::array<Point, 3>;

  // Finally, after all this NECESSARY definitions - code!!!
  Model()
    : m_mesh_ptr_present(),
    m_mesh_ptr_last(),
    m_width(DefModelVal),
```

```
    m_height(DefModelVal),
    m_nodes_x(0),
    m_nodes_y(0),
    m_x_delta(0.0),
    m_y_delta(0.0),
    m_time_delta(0.0) {}

  Model(double width, double height, double delta_n, double time_delta);

  /**
   * Set three points of hole geometry
   * @param p1 first point
   * @param p2 second point
   * @param p3 third point
   */
  void SetHoleGeometry(Point p1, Point p2, Point p3);

  /**
   * Sets the initial conditions of the model
   * @param init_conditions Desired initial condition
   */
  void SetInitialCondition(ModelNodeType init_conditions);

  /*
   * Just the few setters for restrictions, that have not specific behavior.
   */
  void SetOuterRestrictions(
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_up,
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_down,
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_left,
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_right)
   ;
  void SetOuterRestrictions(
    const restr::BoundaryRestrictionsStorageType<ModelNodeType>
    &restrictions);
  void SetInnerRestrictions(
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restriction)
   ;

  void TimeIntegrate(double total_time,
           solution::SolutionStorageBase<ModelNodeType> &storage,
           ModelNodeType tube_flow);
  void SaveResult(solution::SolutionStorageBase<ModelNodeType> &storage)
   const {
  storage.CommitLayer(m_mesh_ptr_present);
  }

private:
  MatrixPointerType m_mesh_ptr_present;
  MatrixPointerType m_mesh_ptr_last;

  double m_width;
  double m_height;
  size_t m_nodes_x;
  size_t m_nodes_y;
  double m_x_delta;
  double m_y_delta;

  double m_time_delta;
```

```
119
120   HoleGeometry m_hole_geometry;
121   restr::BoundaryRestrictionsStorageType<ModelNodeType> m_outer_restrictions
        ;
122   restr::BoundaryRestrincionPointerType<ModelNodeType> m_inner_restriction;
123
124   // Bellow functions helps to determine hole and boundary points related to
125   // hole
126   [[nodiscard]] bool PointInHole(Point point) const;
127   [[nodiscard]] bool PointOnBorder(Point point) const;
128   // Calculates auxiliary values for PointInHole and PointOnBorder function
129   [[nodiscard]] std::tuple<ModelNodeType, ModelNodeType, ModelNodeType>
130   CalcCheckValues(
131     Point point) const;  // sorry for that, it's just a formatter :)))
132   ModelNodeType GetInnerNeighbor(size_t x_shift, size_t y_shift);
133
134   /*
135    * Calculation methods. Just use for improve code readability and
136    * decompose layer calculation.
137    */
138   void ComputeBoundaries();
139   void ComputePlate(ModelNodeType tube_flow);
140 };
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     }
```

```
  }

  return (sign_counter == 3 || sign_counter == 0);
}
}  // anonymous namespace

Model::Model(double width, double height, double delta_n, double time_delta)
    : m_mesh_ptr_present(MatrixBuilder().Build<ModelNodeType>()),
      m_mesh_ptr_last(MatrixBuilder().Build<ModelNodeType>()),
      m_width(width),
      m_height(height),
      m_x_delta(delta_n),
      m_y_delta(delta_n),
      m_time_delta(time_delta) {
  if ((m_time_delta / m_x_delta) * (m_time_delta / m_x_delta) > 0.5) {
    throw exceptions::WrongDeltaRel();
  }
  m_nodes_x = static_cast<size_t>(m_width / m_x_delta);
  m_nodes_y = static_cast<size_t>(m_height / m_y_delta);
  m_mesh_ptr_present->SetSize(m_nodes_y, m_nodes_x);
  m_mesh_ptr_last->SetSize(m_nodes_y, m_nodes_x);

  m_hole_geometry[0] = Point();
  m_hole_geometry[1] = Point();
  m_hole_geometry[2] = Point();
}

void Model::SetHoleGeometry(Point p1, Point p2, Point p3) {
  m_hole_geometry[0] = p1;
  m_hole_geometry[1] = p2;
  m_hole_geometry[2] = p3;
}

void Model::SetInitialCondition(ModelNodeType init_conditions) {
  m_mesh_ptr_present->FillMatrix(init_conditions);
  m_mesh_ptr_last->FillMatrix(init_conditions);
}

void Model::SetOuterRestrictions(
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_up,
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_down,
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_left,
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restr_right)
    {
  m_outer_restrictions[restr::UP_RESTRICTION] = restr_up;
  m_outer_restrictions[restr::DOWN_RESTRICTION] = restr_down;
  m_outer_restrictions[restr::LEFT_RESTRICTION] = restr_left;
  m_outer_restrictions[restr::RIGHT_RESTRICTION] = restr_right;
}

void Model::SetOuterRestrictions(
    const restr::BoundaryRestrictionsStorageType<ModelNodeType> &
   restrictions) {
  m_outer_restrictions = restrictions;
}
void Model::SetInnerRestrictions(
    const restr::BoundaryRestrincionPointerType<ModelNodeType> &restriction)
    {
  m_inner_restriction = restriction;
```

```
}

void Model::TimeIntegrate(double total_time,
                          solution::SolutionStorageBase<ModelNodeType> &
    storage,
                          ModelNodeType tube_flow) {
  storage.CommitLayer(m_mesh_ptr_present);

  auto time_integrate_iterations =
      static_cast<size_t>(total_time / m_time_delta);

  // Iterate time layers
  for (size_t t = 0; t < time_integrate_iterations; ++t) {
    m_mesh_ptr_last = m_mesh_ptr_present;
    ComputeBoundaries();

    ComputePlate(tube_flow);

    storage.CommitLayer(m_mesh_ptr_present);
  }
  storage.CommitLayer(m_mesh_ptr_present);
}

void Model::ComputeBoundaries() {
  // Traverse all boundary nodes necessary

  // Firstly perform left and right boundaries
  for (size_t i = 1; i < m_mesh_ptr_last->SizeRows() - 1; ++i) {
    ModelNodeType T_x_left_inner = m_mesh_ptr_present->GetValue(i, 1);
    ModelNodeType T_x_right_inner =
        m_mesh_ptr_present->GetValue(i, m_mesh_ptr_last->SizeCols() - 2);

    m_mesh_ptr_present->SetValue(
        i, 0,
        m_outer_restrictions[restr::LEFT_RESTRICTION]->operator()(
            T_x_left_inner, m_y_delta));
    m_mesh_ptr_present->SetValue(
        i, m_mesh_ptr_last->SizeCols() - 1,
        m_outer_restrictions[restr::RIGHT_RESTRICTION]->operator()(
            T_x_right_inner, m_y_delta));
  }

  // Finally, perform up and down boundaries
  for (size_t i = 0; i < m_mesh_ptr_last->SizeCols(); ++i) {
    ModelNodeType T_x_down_inner = m_mesh_ptr_present->GetValue(1, i);
    ModelNodeType T_x_up_inner =
        m_mesh_ptr_present->GetValue(m_mesh_ptr_last->SizeRows() - 1, i);
    m_mesh_ptr_present->SetValue(
        0, i,
        m_outer_restrictions[restr::DOWN_RESTRICTION]->operator()(
            T_x_down_inner, m_x_delta));
    m_mesh_ptr_present->SetValue(
        m_mesh_ptr_last->SizeRows() - 1, i,
        m_outer_restrictions[restr::UP_RESTRICTION]->operator()(T_x_up_inner
    ,
                                                                m_x_delta));
  }

  // In fact there is no necessary dependencies for mesh boundary traversal.
```

```
  // So you can change this order if you need.
}

void Model::ComputePlate(ModelNodeType tube_flow) {
  for (size_t j = 1; j < m_mesh_ptr_last->SizeRows() - 1; ++j) {
    for (size_t i = 1; i < m_mesh_ptr_last->SizeCols() - 1; ++i) {
      Point curr_point(static_cast<double>(i) * m_x_delta,
                       static_cast<double>(j) * m_y_delta);
      if (!PointInHole(curr_point)) {
        if (PointOnBorder(curr_point)) {
          ModelNodeType inner_value = GetInnerNeighbor(i, j);
          m_mesh_ptr_present->SetValue(
              j, i, m_inner_restriction->operator()(inner_value, m_x_delta))
    ;
        } else {
          equations::HeatConductionParamsType<ModelNodeType>
              equation_parameters{m_mesh_ptr_last->GetValue(j, i),
                                  m_mesh_ptr_last->GetValue(j, i - 1),
                                  m_mesh_ptr_last->GetValue(j, i + 1),
                                  m_mesh_ptr_last->GetValue(j - 1, i),
                                  m_mesh_ptr_last->GetValue(j + 1, i),
                                  m_time_delta,
                                  m_x_delta,
                                  m_y_delta,
                                  0.1};
          m_mesh_ptr_present->SetValue(
              j, i, equations::HeatConductionProblem(equation_parameters));
        }
      } else {
        m_mesh_ptr_present->SetValue(j, i,
                                     std::forward<ModelNodeType>(tube_flow))
    ;
      }
    }
  }
}

std::tuple<Model::ModelNodeType, Model::ModelNodeType, Model::ModelNodeType>
Model::CalcCheckValues(Point point) const {
  ModelNodeType check_val1 = (m_hole_geometry[0].x - point.x) *
                                 (m_hole_geometry[1].y - m_hole_geometry[0].
    y) -
                             (m_hole_geometry[1].x - m_hole_geometry[0].x) *
                                 (m_hole_geometry[0].y - point.y);
  ModelNodeType check_val2 = (m_hole_geometry[1].x - point.x) *
                                 (m_hole_geometry[2].y - m_hole_geometry[1].
    y) -
                             (m_hole_geometry[2].x - m_hole_geometry[1].x) *
                                 (m_hole_geometry[1].y - point.y);
  ModelNodeType check_val3 = (m_hole_geometry[2].x - point.x) *
                                 (m_hole_geometry[0].y - m_hole_geometry[2].
    y) -
                             (m_hole_geometry[0].x - m_hole_geometry[2].x) *
                                 (m_hole_geometry[2].y - point.y);
  return {check_val1, check_val2, check_val3};
}

Model::ModelNodeType Model::GetInnerNeighbor(size_t x_shift, size_t y_shift)
    {
```

```
183   auto cast_x_shift = static_cast<double>(x_shift);
184   auto cast_y_shift = static_cast<double>(y_shift);
185   Point up_neighbor((cast_x_shift + 1) * m_x_delta, cast_y_shift * m_y_delta
       );
186   Point down_neighbor((cast_x_shift - 1) * m_x_delta, cast_y_shift *
       m_y_delta);
187   Point right_neighbor(cast_x_shift * m_x_delta,
188                        (cast_y_shift - 1) * m_y_delta);
189   Point left_neighbor(cast_x_shift * m_x_delta, (cast_y_shift + 1) *
       m_y_delta);
190
191   if (!PointOnBorder(up_neighbor) && !PointInHole(up_neighbor)) {
192     return m_mesh_ptr_last->GetValue(y_shift, x_shift + 1);
193   }
194
195   if (!PointOnBorder(down_neighbor) && !PointInHole(down_neighbor)) {
196     return m_mesh_ptr_last->GetValue(y_shift, x_shift - 1);
197   }
198
199   if (!PointOnBorder(right_neighbor) && !PointInHole(right_neighbor)) {
200     return m_mesh_ptr_last->GetValue(y_shift + 1, x_shift);
201   }
202
203   if (!PointOnBorder(left_neighbor) && !PointInHole(left_neighbor)) {
204     return m_mesh_ptr_last->GetValue(y_shift - 1, x_shift);
205   }
206
207   return 0.0;
208 }
209
210 bool Model::PointInHole(Point point) const {
211   /*
212    * Mathematical part - vector and pseudoscalar product.
213    * Implementation - products are considered (1,2,3 - triangle vertices, 0
       -
214    * point):
215    * (x1-x0)*(y2-y1)-(x2-x1)*(y1-y0)
216    * (x2-x0)*(y3-y2)-(x3-x2)*(y2-y0)
217    * (x3-x0)*(y1-y3)-(x1-x3)*(y3-y0)
218    * If they are of the same sign, then the point is inside
219    * the triangle, otherwise the point is outside the triangle.
220    */
221   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<
```

```
        fdm::restr::SecondKindRestriction<fdm::Model::ModelNodeType>>(40.0);
    restrictions[fdm::restr::LEFT_RESTRICTION] = std::make_shared<
        fdm::restr::SecondKindRestriction<fdm::Model::ModelNodeType>>(40.0);
    restrictions[fdm::restr::RIGHT_RESTRICTION] = std::make_shared<
        fdm::restr::SecondKindRestriction<fdm::Model::ModelNodeType>>(40.0);
    fdm::restr::BoundaryRestrincionPointerType<fdm::Model::ModelNodeType>
        inner_restr = std::make_shared<
            fdm::restr::ThirdKindRestriction<fdm::Model::ModelNodeType>>();

    model.SetOuterRestrictions(restrictions);
    model.SetInnerRestrictions(inner_restr);
    model.SetHoleGeometry(fdm::Model::Point(2.0, 1.0),
                          fdm::Model::Point(5.0, 1.0),
                          fdm::Model::Point(5.0, 3.0));

    fdm::solution::PlaceholderStorage<fdm::Model::ModelNodeType>
        stream_storage_placeholder;
    fdm::solution::StaticGnuplotHeatmapStorage<fdm::Model::ModelNodeType>
      stream_storage_gnuplot("heatmap");

    model.TimeIntegrate(25.0, stream_storage_placeholder, 0);
    model.SaveResult(stream_storage_gnuplot);
    return 0;
}
```

Листинг 6: solution.cpp