



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 *Робототехники и комплексной автоматизации*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

по дисциплине: «Модели и методы анализа проектных решений»

Студент	Тетерин Никита Евгеньевич
Группа	РК6-64Б
Тип задания	Вариант 79
Тема лабораторной работы	Метод конечных разностей для решения задачи теплопроводности

Студент	_____ <u>Тетерин Н.Е.</u>
	<i>подпись, дата</i> <i>фамилия, и.о.</i>
Преподаватель	_____ <u>Трудоношин В.А.</u>
	<i>подпись, дата</i> <i>фамилия, и.о.</i>

Оценка _____

Москва, 2022 г.

Оглавление

Задание на лабораторную работу	3
Теоретическая часть.....	4
Описание структуры программы	7
Пример работы программы	8
Исходный код программы	11

Задание на лабораторную работу

Вариант 79.

С помощью неявной разностной схемы решить нестационарное уравнение теплопроводности для пластины, изображенной на рис. 1., там же указаны габариты пластины. Начальное значение температуры пластины – 0 градусов. Граничные условия, следующие: слева теплоизолировано, внизу 50 градусов, на верхней границе и справа на наклонной границе 80 градусов. На внутренней границе отверстия $dT/dn = T$. При выводе результатов показать динамику изменения температуры (например, с помощью цветовой гаммы).

Теоретическая часть

Нестационарная задача представляет собой задачу, в которой определяется изменение поля фазовой переменной во времени. Нестационарное уравнение теплопроводности в упрощенной форме может быть записано следующим образом:

$$\frac{\partial T}{\partial t} = k \left(\frac{\partial T^2}{\partial x^2} + \frac{\partial T^2}{\partial y^2} \right)$$

Численное решение ДУ предполагает в первую очередь переход от производных к конечным разностям (численное дифференцирование) с дальнейшим применением некоторой численной схемы для интегрирования во времени, будь то метод Эйлера, Рунге-Кутты и т. д. Неявная численная схема (неявный метод Эйлера) в общем случае предполагает решение СНАУ относительно вектора очередных состояний T , однако при реализации применительно к конкретному ДУ можно преобразовать СНАУ к СЛАУ путем переноса членов, включающих независимые переменные, в левую часть уравнения. Далее подобная система может быть решена традиционными методами для решения СЛАУ.

В данной работе для решения ДУ использовалась т. н. разностная схема расщепления. Она представляет собой переход от двумерной задачи к двум одномерным следующим образом:

$$\frac{\partial V}{\partial t} = k \frac{\partial T^2}{\partial x^2}, \quad \frac{\partial W}{\partial t} = k \frac{\partial T^2}{\partial y^2}$$

$$V(x, y, t_i) = T(x, y, t_i), W(x, y, t_i) = V(x, y, t_{i+1})$$

$$W(x, y, t_{i+1}) \approx T(x, y, t_{i+1}) + \Theta(\Delta t_i^2)$$

Неявную разностную схему расщепления можно представить следующим образом:

$$\frac{V_{x,y}^{k+1} - V_{x,y}^k}{\Delta t} = k \frac{V_{x+1,y}^{k+1} - 2V_{x,y}^{k+1} + V_{x-1,y}^{k+1}}{\Delta x^2}$$

$$\frac{W_{x,y}^{k+1} - W_{x,y}^k}{\Delta t} = k \frac{W_{x,y+1}^{k+1} - 2W_{x,y}^{k+1} + W_{x,y-1}^{k+1}}{\Delta y^2}$$

Таким образом, значение фазовой переменной может быть аппроксимировано путем решения последовательно двух одномерных задач, при этом

$$V_{x,y}^k = T_{x,y}^k, W_{x,y}^k = V_{x,y}^{k+1}, T_{x,y}^{k+1} = W_{x,y}^{k+1}$$

Данное преобразование является выгодным в том числе и потому, что результирующая матрица коэффициентов является ленточной с шириной ленты, равной 3. Для решения СЛАУ с подобной матрицей можно применить метод прогонки (tridiagonal matrix algorithm, Thompson algorithm), который является линейным относительно размерности матрицы в смысле количества производимых операций, к тому же эффективен в смысле потребляемой памяти, ведь из всей матрицы необходимо в каждый момент времени хранить только 3 диагонали.

Введенные в условия задачи граничные условия представляют собой следующее: ГУ 1 рода (условия Дирихле) – значение фазовой переменной является постоянным: $T = const$. ГУ 2 рода (условия Неймана) предполагают постоянное значение потока фазовой переменной вдоль направления (в нашем случае, по горизонтали): $\frac{\partial T}{\partial n} = const$. ГУ 3 рода определяют уравнения баланса, в общем случае ДУ вида $f\left(\frac{\partial T}{\partial n}, T\right) = 0$. В случае задач теплопроводности это уравнение теплового баланса: тепловой поток с границы объекта зависит от температуры). Так, в данной задаче данное ГУ формулировалось в виде:

$$\frac{\Delta T}{\Delta n} = T$$

Для метода прогонки необходимо записать выражения для конечных разностей в узлах:

$$\begin{aligned}\frac{T_{x,y}^{i+1} - T_{x,y}^i}{\Delta t} &= k \frac{T_{x+1,y}^{i+1} - 2T_{x,y}^{i+1} + T_{x-1,y}^{i+1}}{\Delta x^2} \Rightarrow \\ T_{x,y}^{i+1} - T_{x,y}^i &= \frac{k\Delta t}{\Delta x^2} (T_{x+1,y}^{i+1} - 2T_{x,y}^{i+1} + T_{x-1,y}^{i+1}) \Rightarrow \\ r = \frac{k\Delta t}{\Delta x^2} &\Rightarrow -rT_{x+1,y}^{i+1} + (1 + 2r)T_{x,y}^{i+1} - rT_{x-1,y}^{i+1} = T_{x,y}^i \Rightarrow \\ [-r \quad 2r + 1 \quad -r] &\begin{bmatrix} T_{x-1,y}^{i+1} \\ T_{x,y}^{i+1} \\ T_{x+1,y}^{i+1} \end{bmatrix} = T_{x,y}^i\end{aligned}$$

Общая матрица коэффициентов для замкнутой СЛАУ получается путем ансамблирования подобных систем из 3 уравнений и добавления строк, соответствующих узлам, в которых заданы ГУ. Для узлов, в которых заданы ГУ 1 рода, строка имеет 1 единицу на главной диагонали, при этом все остальные вхождения равны 0. Для узлов, в которых определены ГУ 2 рода, строка имеет 1 на главной диагонали и -1 в соседнем узле, при этом соответствующее вхождение в векторе свободных членов равно константному значению производной (потока), в данном случае, 0 (граница тела теплоизолированная). Для узлов, в которых определены ГУ 3 рода, можно записать:

$$\frac{T_{n-1} - T_n}{\Delta n} = T_n \Rightarrow T_n - \frac{T_{n-1}}{1 + \Delta n} = 0 \Rightarrow \begin{bmatrix} 1 & -\frac{1}{1 + \Delta n} \end{bmatrix} \begin{bmatrix} T_n \\ T_{n-1} \end{bmatrix} = 0$$

При этом удобно полагать, что тепловое излучение всегда направлено из тела для единообразия выражений.

Метод прогонки представляет собой модификацию метода Гаусса с учетом специфики ленточной матрицы, поскольку каждая строка имеет не более 3 ненулевых элемента, расположенных вокруг главной диагонали (кроме крайних, на которых находится не более 2 ненулевых элементов). Как и в методе Гаусса, метод включает в себя прямой и обратный ход. Во время прямого хода матрица приводится к верхнему треугольному виду, а во время обратного итеративно определяются значения вектора неизвестных.

Описание структуры программы

Программа состоит из 3 основных компонент: сущность геометрической модели, которая поддерживает интерфейс модели, т. е. может возвращать коэффициенты ленточной матрицы вдоль каждой из осей (x и y) для каждого внутреннего узла сетки, а также для всех граничных узлов и значения вектора свободных членов, скрывая в себе логику, отвечающую за ГУ. Следующая сущность – solver, решатель, реализующий метод прогонки. Сущность GNUPlotWriter оборачивает вызов и IO для утилиты gnuplot. Сущность Problem оборачивает модель и решатели. Она имеет метод step, который осуществляет шаг по времени. Его необходимо вызывать необходимое число раз, при этом на каждой итерации необходимо делать дампы состояния поля температур в gnuplot.

Основная идея программы заключается в том, что решатель для метода прогонки и конкретная конфигурация системы (совокупность геометрии и ГУ) являются независимыми компонентами. Так, возможно добавить другой класс модели, реализующий интерфейс IModel, при этом остальная часть программы не нуждается в модификации. Кроме того, в данной реализации метод прогонки для решения набора одномерных задач применяется последовательно, в то время как, вообще говоря, данные подзадачи являются независимыми и могут исполняться параллельно. Также стоит отметить, что решение нестационарных задач большой размерности (в смысле числа узлов

сетки) для большого числа временных слоев может приводить к огромным затратам ОЗУ, поэтому эту задачу надо производить в потоковом режиме, т. е. в каждый момент времени модель хранит только 1 матрицу с полем температур.

Сетка в данной задаче должна быть достаточно мелкой, чтобы вмещать узлы с дробными координатами, такие как границы отверстия в пластине (значения ординат узлов на верхней и нижней стенках отверстия равны соответственно 3.5 и 1.5). Для учета типов граничных условий в пространстве имен `model` создано перечисление `condition`, которое содержит ГУ 1, 2 и 3 родов вдоль каждой из осей, а также ГУ 3 рода для обеих осей, которое возникает в угловых узлах прямоугольного отверстия. Кроме того, определены типы внешних и внутренних узлов без ГУ. Очевидно, во время вычислений температурное поле вне пластины считается стационарным.

Пример работы программы

В начале работы программа выводит в терминал условно-схематическое изображение полученной сетки, на котором можно наблюдать различные типы ГУ. Пример такого вывода приведен на рис.1. На рис. 2 приведены НУ системы, а на рис. 3 – результат работы программы: поле температур в момент времени 15с. Для сравнения на рис. 4 приведен результат, полученный для такой же системы в ANSYS. Можно наблюдать, что (с точностью до перевернутого вывода) решения, полученные обоими способами, совпадают. Из этого можно заключить, что численное решение ДУ и метод прогонки реализованы корректно.

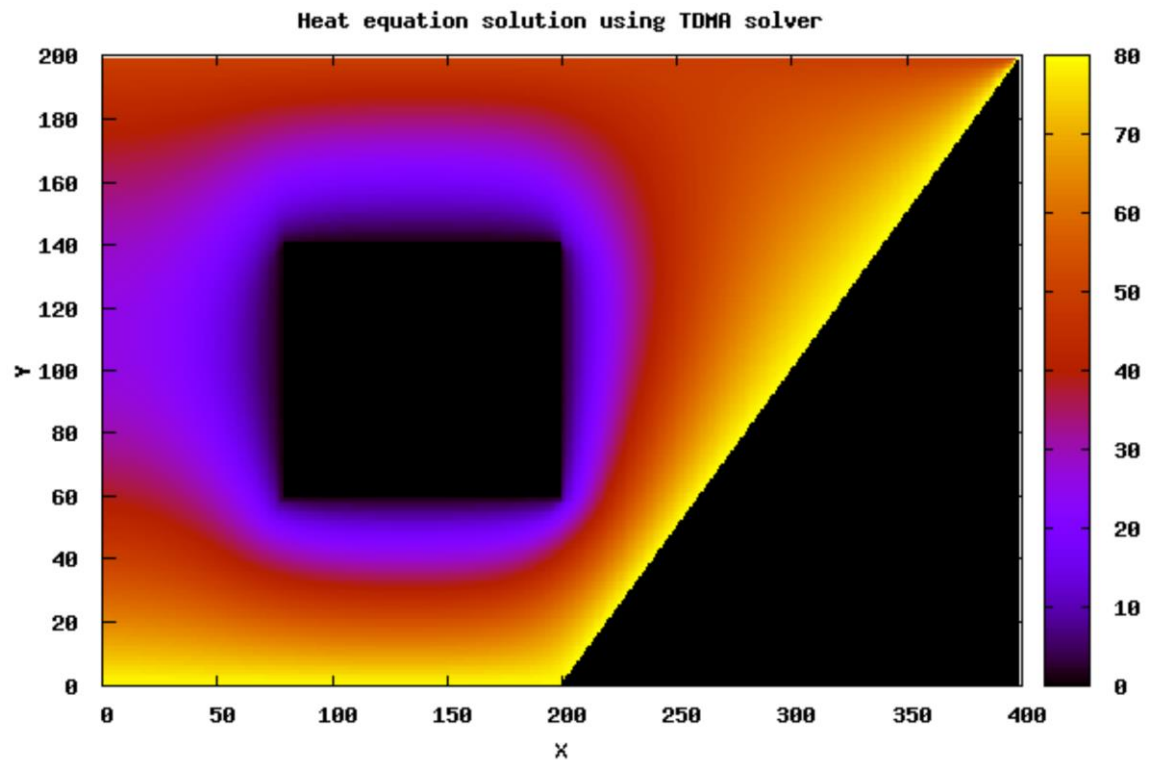


Рис. 3. Поле температур в момент времени 15с. (сетка 400x200, 1000 шагов метода).

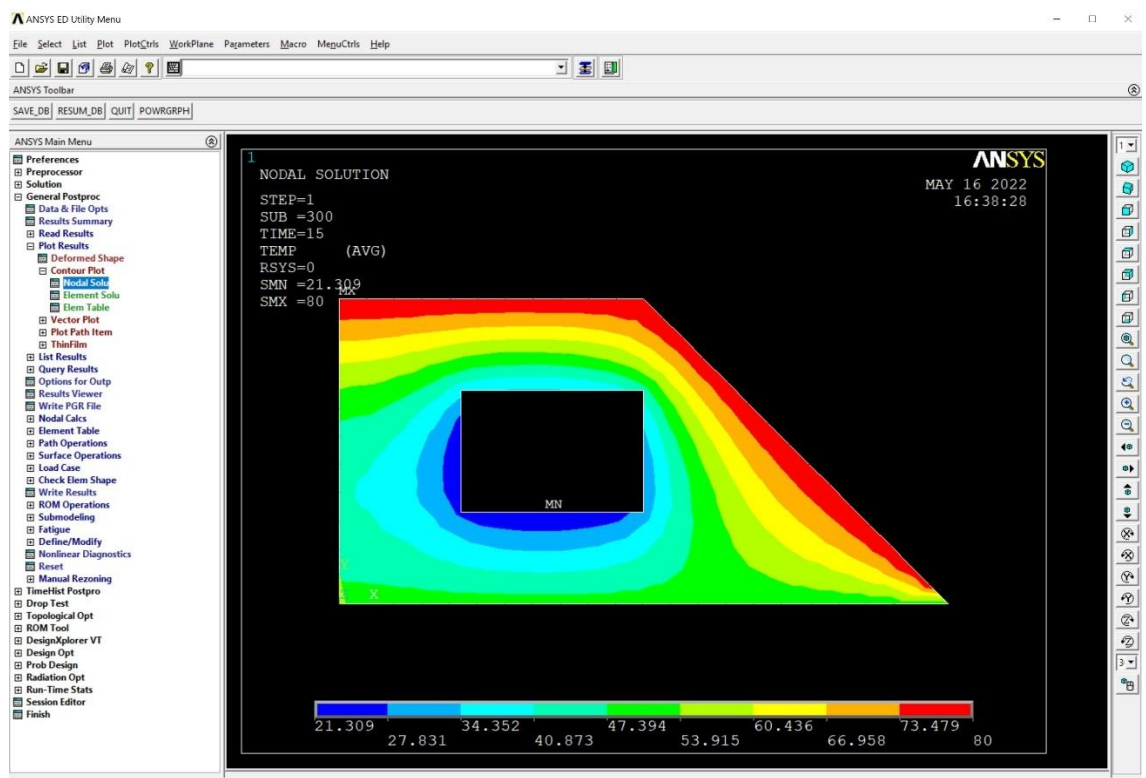


Рис. 4. Результат моделирования этой же системы в ANSYS.


```
sakeof@LAPTOP-VN7C63DA:~/mess/mmapr$ cat compile.sh
mkdir -p build
cd build
cmake ..
make
cd -
sakeof@LAPTOP-VN7C63DA:~/mess/mmapr$
sakeof@LAPTOP-VN7C63DA:~/mess/mmapr$
```

Рис. 7. Компиляция проекта.

```
sakeof@LAPTOP-VN7C63DA:~/mess/mmapr$ tree -I build
.
├── CMakeLists.txt
├── README.md
├── docs
└── project
    ├── CMakeLists.txt
    ├── include
    │   ├── model.hpp
    │   ├── plotter.hpp
    │   ├── shared.hpp
    │   └── solver.hpp
    ├── main.cpp
    └── source
        ├── model.cpp
        ├── plotter.cpp
        └── solver.cpp

4 directories, 11 files
sakeof@LAPTOP-VN7C63DA:~/mess/mmapr$
```

Рис. 8. Файловая структура проекта.

main.c

```
#include <iostream>
#include <memory>

#include "solver.hpp"
#include "plotter.hpp"

constexpr size_t DEF_Timesteps = 1000;
constexpr double DEF_TIME = 15.0;
constexpr size_t X_NODES = 200;
constexpr size_t Y_NODES = 100;
constexpr double X_LEN = 10.0;
constexpr double Y_LEN = 5.0;

constexpr std::string_view running = "Performing computations: ";
constexpr std::string_view usage = "Usage: <simulation time:double>
[<timesteps:uint> [<x_nodes:uint> <y_nodes:uint>]]\n";

int main(int argc, char* argv[]) {
    double time = DEF_TIME;
    double timesteps = DEF_Timesteps;
```

```

const double a = 1.0;

size_t x_nodes = X_NODES;
size_t y_nodes = Y_NODES;

// not the most versatile solution, however
// it is OK for this case
if (argc == 1) {
    std::cout << usage;
    return EXIT_FAILURE;
}
if (argc >= 2) {
    time = std::stod(argv[1]);
}
if (argc >= 3) {
    timesteps = std::stoul(argv[2]);
}
if (argc == 5) {
    x_nodes = std::stoul(argv[3]);
    y_nodes = std::stoul(argv[4]);

    if (x_nodes != y_nodes * 2) {
        std::cerr << "X to Y ratio must be 2:1\n";
        return EXIT_FAILURE;
    }
}

std::cout << "Simulation time set to " << time << '\n';
std::cout << "Timesteps set to " << timesteps << '\n';
std::cout << "Mesh size: [" << x_nodes << ':' << y_nodes << "]\n";

const double dt = time / static_cast<double>(timesteps);
const double dx = X_LEN / static_cast<double>(x_nodes);
const double dy = Y_LEN / static_cast<double>(y_nodes);

// instantiate model for my case, set up problem
// environment (e.g., allocate memory for solvers)
// and gnuplot wrapper to create heatmap gif
model::Model79 m(dt, dx, dy, a, x_nodes, y_nodes);
solver::Problem problem(m, timesteps);
plt::GNUPlotWriter plotter(plt::GNUPlotWriter::basic_gif_config.data());

std::cout << "The problem schematic (may not fit into the terminal
entirely)\n";
pprint_grid(m, std::cout);
plotter.receiver() << m;
plotter.flush_buffer();

std::cout << running;
for (size_t i = 0; i < timesteps; ++i) {

```

```

        std::cout << '\r' << running << "iter [" << i << '/' << timesteps << ']';
        std::cout.flush();

        problem.step();
        plotter.reciever() << m;
        plotter.flush_buffer();
    }

    std::cout << " Done, OK\n";
    return EXIT_SUCCESS;
}

```

shared.hpp

```

#pragma once

#include <vector>
#include <array>

```

model.hpp

```

#pragma once

#include "shared.hpp"

#define T_FLOOR 50.0
#define T_CEIL 80.0

namespace model {
    using tridiag_coefs = std::array<double, 3>;
    using boundary_coefs = std::array<double, 2>;

    enum condition {
        NO_CONDITION,
        BOUNDARY_1TYPE,
        BOUNDARY_2TYPE_X,
        BOUNDARY_2TYPE_Y,
        BOUNDARY_3TYPE_X,
        BOUNDARY_3TYPE_Y,
        BOUNDARY_3TYPE_XY,
        OUTER_NODE
    };

    enum direction {
        VERTICAL,
        HORIZONTAL
    };
}

```

```

};

// model instance should provide sets of coefficients
// for each grid point in horizontal and vertical directions
// to apply the Thompson's tridiagonal matrix algorithm
// additionally, model can store required parameters like dt, dx, dy
// and a_1, a_2
// the heat PDE is defined as follows:
//  $\frac{\partial T}{\partial t} = a_1 * \frac{\partial^2 T}{\partial x^2} + a_2 * \frac{\partial^2 T}{\partial y^2}$ 
class IModel {
protected:
    virtual void dump(std::ostream & os) const = 0;
public:
    virtual void set_current_value(const size_t x, const size_t y, const
double value) = 0;
    virtual double get_RHS_coefs_x(const size_t x, const size_t y) const = 0;
    virtual double get_RHS_coefs_y(const size_t x, const size_t y) const = 0;
    virtual tridiag_coefs get_x_coefs(const size_t x, const size_t y) const =
0;
    virtual tridiag_coefs get_y_coefs(const size_t x, const size_t y) const =
0;
    virtual boundary_coefs get_x_last_coefs(const size_t y) const = 0;
    virtual boundary_coefs get_y_last_coefs(const size_t y) const = 0;
    virtual boundary_coefs get_x_first_coefs(const size_t x) const = 0;
    virtual boundary_coefs get_y_first_coefs(const size_t x) const = 0;
    virtual size_t x_dim() const = 0;
    virtual size_t y_dim() const = 0;
    friend std::ostream & operator<<(std::ostream & os, const IModel & m) {
        m.dump(os);
        return os;
    }
};

// Model79 implements IModel interface and stands for my particular problem
setup
// thus such methods as is_inner and is_border are present to deduce
// the geometry. This is not the most elegant approach, however...
class Model79: public IModel {
protected:
    void dump(std::ostream & os) const override;
private:
    // holds boundary condition type if present
    // and the default temperature value in case of
    // Dirichlet's BC (T = const)
    struct Node {
        condition condition_type = NO_CONDITION;
        double current_value = 0;
        double initial_value = 0;
    };
};
// Grid entity contains matrix of condition flags

```



```

// for ease boundary condition detection
struct Grid {
    const size_t width;
    const size_t height;
    std::vector<std::vector<Node>> nodes;
    Grid(const size_t width, const size_t height):
        width(width), height(height),
        nodes(height, std::vector<Node>(width)) {};
    ~Grid() = default;
};

private:
    const double dt;
    const double dx;
    const double dy;
    const double a;
    const std::pair<size_t, size_t> dims;
    Grid grid;

private:
    void throw_on_bounds(const size_t x, const size_t y) const;
    void grid_set_up(); // init grid with required flags + default values
    bool is_inner(const size_t x, const size_t y) const;

public:
    virtual void set_current_value(const size_t x, const size_t y, const
double value) override;
    double get_RHS_coefs_x(const size_t x, const size_t y) const override;
    double get_RHS_coefs_y(const size_t x, const size_t y) const override;
    tridiag_coefs get_x_coefs(const size_t x, const size_t y) const override;
    tridiag_coefs get_y_coefs(const size_t x, const size_t y) const override;
    boundary_coefs get_x_last_coefs(const size_t y) const override;
    boundary_coefs get_y_last_coefs(const size_t y) const override;
    boundary_coefs get_x_first_coefs(const size_t x) const override;
    boundary_coefs get_y_first_coefs(const size_t x) const override;
    size_t x_dim() const override { return dims.first; }
    size_t y_dim() const override { return dims.second; }

~Model79() = default;
Model79() = delete;
Model79(
    const double dt,
    const double dx,
    const double dy,
    const double a,
    const size_t x_nodes,
    const size_t y_nodes) :
    dt(dt), dx(dx), dy(dy), a(a),
    dims(std::make_pair(x_nodes, y_nodes)),
    grid(x_nodes, y_nodes) { grid_set_up(); };

    friend void pprint_grid(const Model79 & m, std::ostream & out);
};
}

```


model.cpp[illegible]


```

// left side
for (size_t i = 0; i < y_dim; ++i) {
    grid.nodes[i][0].condition_type = BOUNDARY_2TYPE_X;
}

// floor
for (size_t i = 0; i < x_dim; ++i) {
    grid.nodes[y_dim - 1][i].condition_type = BOUNDARY_1TYPE;
    grid.nodes[y_dim - 1][i].current_value = T_FLOOR;
    grid.nodes[y_dim - 1][i].initial_value = T_FLOOR;
}

// ceiling
for (size_t i = 0; i < x_half; ++i) {
    grid.nodes[0][i].condition_type = BOUNDARY_1TYPE;
    grid.nodes[0][i].current_value = T_CEIL;
    grid.nodes[0][i].initial_value = T_CEIL;
}

// right side
for (size_t i = 0; i < x_half; ++i) {
    // nodes to the right to the inclined side
    for (size_t j = i; j < y_dim; ++j) {
        grid.nodes[y_dim - j - 1][x_dim - i - 1].condition_type =
OUTER_NODE;
    }
    grid.nodes[x_half - i - 1][x_dim - i - 1].condition_type =
BOUNDARY_1TYPE;
    grid.nodes[x_half - i - 1][x_dim - i - 1].current_value = T_CEIL;
    grid.nodes[x_half - i - 1][x_dim - i - 1].initial_value = T_CEIL;
}

// hole
for (size_t i = x_hole_left + 1; i < x_hole_right; ++i) {
    for (size_t j = y_hole_lower + 1; j < y_hole_upper; ++j) {
        // nodes inside the hole are marked as outer
        grid.nodes[j][i].condition_type = OUTER_NODE;
    }
}
for (size_t i = x_hole_left + 1; i < x_hole_right; ++i) {
    grid.nodes[y_hole_lower][i].condition_type = BOUNDARY_3TYPE_Y;
    grid.nodes[y_hole_upper][i].condition_type = BOUNDARY_3TYPE_Y;
}
for (size_t i = y_hole_lower + 1; i < y_hole_upper; ++i) {
    grid.nodes[i][x_hole_left].condition_type = BOUNDARY_3TYPE_X;
    grid.nodes[i][x_hole_right].condition_type = BOUNDARY_3TYPE_X;
}

// corner nodes
grid.nodes[y_hole_upper][x_hole_left].condition_type = BOUNDARY_3TYPE_XY;

```

```

        grid.nodes[y_hole_upper][x_hole_right].condition_type =
BOUNDARY_3TYPE_XY;
        grid.nodes[y_hole_lower][x_hole_right].condition_type =
BOUNDARY_3TYPE_XY;
        grid.nodes[y_hole_lower][x_hole_left].condition_type = BOUNDARY_3TYPE_XY;
    }

    bool Model79::is_inner(const size_t x, const size_t y) const {
        throw_on_bounds(x, y);
        return not (grid.nodes[y][x].condition_type == OUTER_NODE);
    }

    void Model79::set_current_value(const size_t x, const size_t y, const double
value) {
        throw_on_bounds(x, y);
        const condition cond = grid.nodes[y][x].condition_type;
        if (cond == BOUNDARY_1TYPE or cond == OUTER_NODE) return;
        grid.nodes[y][x].current_value = value;
    }

    double Model79::get_RHS_coefs_x(const size_t x, const size_t y) const {
        throw_on_bounds(x, y);
        const Node grid_node = grid.nodes[y][x];

        switch (grid_node.condition_type) {
            case OUTER_NODE:
                return grid_node.initial_value;
            case BOUNDARY_2TYPE_X:
            case BOUNDARY_3TYPE_X:
                return 0;
            case BOUNDARY_3TYPE_XY:
            case BOUNDARY_2TYPE_Y:
            case BOUNDARY_3TYPE_Y:
            case BOUNDARY_1TYPE:
            case NO_CONDITION:
                return grid_node.current_value;
            default:
                throw std::runtime_error("unknown condition type");
        }
    }

    double Model79::get_RHS_coefs_y(const size_t x, const size_t y) const {
        throw_on_bounds(x, y);
        const Node grid_node = grid.nodes[y][x];

        switch (grid_node.condition_type) {
            case OUTER_NODE:
                return grid_node.initial_value;
            case BOUNDARY_2TYPE_Y:
            case BOUNDARY_3TYPE_Y:
                return 0;

```

```

        case BOUNDARY_3TYPE_XY:
        case BOUNDARY_2TYPE_X:
        case BOUNDARY_3TYPE_X:
        case BOUNDARY_1TYPE:
        case NO_CONDITION:
            return grid_node.current_value;
        default:
            throw std::runtime_error("unknown condition type");
    }
}

tridiag_coefs Model79::get_x_coefs(const size_t x, const size_t y) const {
    throw_on_bounds(x, y);
    const double R = a * dt / (dx * dx);    // needed for nodes with no
boundary
    const condition cond = grid.nodes[y][x].condition_type;

    switch (cond) {
        // must be constant value
        case OUTER_NODE:
        case BOUNDARY_1TYPE:
            return {0, 1.0, 0};
        case BOUNDARY_3TYPE_XY:
        case BOUNDARY_3TYPE_X: {
            if (grid.nodes[y][x + 1].condition_type == NO_CONDITION) {
                const double c = (-1.0) / (1.0 + dx);
                return {c, 1.0, 0};
            }
            if (grid.nodes[y][x - 1].condition_type == NO_CONDITION) {
                const double c = (-1.0) / (1.0 + dx);
                return {0, 1.0, c};
            }
        }
        case BOUNDARY_3TYPE_Y:
        case BOUNDARY_2TYPE_Y:
        case NO_CONDITION: {
            return {-R, 2.0*R + 1.0, -R};
        }
        default:
            throw std::runtime_error("unknown condition type");
    }
}

tridiag_coefs Model79::get_y_coefs(const size_t x, const size_t y) const {
    throw_on_bounds(x, y);
    const double R = a * dt / (dy * dy);
    const condition cond = grid.nodes[y][x].condition_type;

    switch (cond) {
        // must be constant value
        case OUTER_NODE:

```

```

    case BOUNDARY_1TYPE:
        return {0, 1.0, 0};
    case BOUNDARY_3TYPE_XY:
    case BOUNDARY_3TYPE_Y: {
        // in this particular case, 3rd type boundaries
        // are only defined at inner nodes -> if one has these at the
edge
        // of the mesh, it is a good idea to bound-check the axis first
        // in order to avoid segfaults
        if (grid.nodes[y + 1][x].condition_type == NO_CONDITION) {
            const double c = (-1.0) / (1.0 + dx);
            return {c, 1.0, 0};
        }
        if (grid.nodes[y - 1][x].condition_type == NO_CONDITION) {
            const double c = (-1.0) / (1.0 + dx);
            return {0, 1.0, c};
        }
    }
    // in this particular problem, 2nd type boundaries
    // only appears on the edge
    // in X-direction thus are treated as ordinary inner nodes
    case BOUNDARY_3TYPE_X:
    case BOUNDARY_2TYPE_X:
    case NO_CONDITION: {
        return {-R, 2.0*R + 1.0, -R};
    }
    default:
        throw std::runtime_error("unknown condition type");
}

}

// numeration order is as follows:
// upmost nodes = 0, lowest nodes = x_max, leftmost nodes = 0,
rightmost_nodes = y_max
// these functions would only apply to my particular problem
boundary_coefs Model79::get_x_last_coefs(const size_t x) const {
    throw_on_bounds(x, 0);
    return {0, 1.0};
}

boundary_coefs Model79::get_x_first_coefs(const size_t y) const {
    throw_on_bounds(0, y);
    // in this case, the 2 TYPE boundary is defined on the
    // leftmost side of the plate
    return {-1.0, 1.0};
}

boundary_coefs Model79::get_y_last_coefs(const size_t x) const {
    throw_on_bounds(x, 0);
    return {0.0, 1.0};
}

```

```

boundary_coefs Model79::get_y_first_coefs(const size_t x) const {
    throw_on_bounds(x, 0);
    return {1.0, 0.0};
}

static char cond_to_symbol(const condition c) {
    switch (c) {
        case NO_CONDITION:
            return '#';
        case BOUNDARY_1TYPE:
            return '*';
        case BOUNDARY_2TYPE_X:
        case BOUNDARY_2TYPE_Y:
            return '@';
        case BOUNDARY_3TYPE_X:
            return '-';
        case BOUNDARY_3TYPE_Y:
            return '|';
        case BOUNDARY_3TYPE_XY:
            return 'x';
        case OUTER_NODE:
            return '.';
        default:
            throw std::runtime_error("unknown condition type");
    }
}

void pprint_grid(const Model79 & m, std::ostream & out) {
    const size_t x_dim = m.x_dim();
    const size_t y_dim = m.y_dim();
    for (size_t i = 0; i < y_dim; ++i) {
        for (size_t j = 0; j < x_dim; ++j) {
            out << cond_to_symbol(m.grid.nodes[i][j].condition_type) << ' ';
        }
        out << '\n';
    }
}
}

```

solver.hpp

```

#pragma once

#include "model.hpp"

namespace solver {
    using diagonal = std::vector<double>;
    using tridiagonal_mx_extended = std::array<diagonal, 4>;
}

```

```

// TDMA stands for tridiagonal matrix algorithm
// aka Thomas algorithm in en literature
// this one can solve sets of linear equations (SLEs)
// with tridiagonal matrix in linear time
class TDMA {
private:
    diagonal c_star;
    diagonal d_star;
private:
public:
    TDMA() = default;
    TDMA(const size_t diagonal_length);
    ~TDMA() = default;
    void solve(const tridiagonal_mx_extended & newSLE, diagonal & storage);
};

// Problem entity wraps everything, i.e. the model and solvers
// (also allocates some auxiliary storage once for the run)
// problem is solved in an iterative manner, with grid's current
// values updated at each step
class Problem {
private:
    size_t current_step = 0;
    const size_t n_iters = 0;
    model::IModel & m;
    TDMA solver_x;
    TDMA solver_y;
    tridiagonal_mx_extended mx_x;
    tridiagonal_mx_extended mx_y;
    diagonal f_x;
    diagonal f_y;
private:
    void update_grid_row(const size_t y);
    void update_grid_col(const size_t x);
public:
    Problem() = delete;
    Problem(model::IModel & model, const size_t n_iters);
    void step();
};
}

```

solver.cpp

```

#include "solver.hpp"

#include <algorithm>
#include <iostream>

constexpr bool VERBOSE = false;

namespace solver {

```



```

TDMA::TDMA(const size_t diagonal_length):
c_star(diagonal_length, 0), d_star(diagonal_length, 0) {}

void TDMA::solve(
    const tridiagonal_mx_extended & SLE,
    diagonal & storage
) {
    const diagonal a = SLE[0], b = SLE[1], c = SLE[2], d = SLE[3];
    const size_t N = b.size();

    if (N != a.size())
        throw std::runtime_error("dimension mismatch for a");
    if (N != c.size())
        throw std::runtime_error("dimension mismatch for c");
    if (N != d.size())
        throw std::runtime_error("dimension mismatch for d");
    if (N != storage.size())
        throw std::runtime_error("dimension mismatch for storage");

    // reallocate memory for auxiliary
    // collections if the dimension has changed
    if (c_star.size() != N) {
        std::cerr
        << "changes c^* size: was "
        << c_star.size()
        << " now: "
        << N;
        c_star = diagonal(N, 0);
        d_star = diagonal(N, 0);
    }

    // update the coefficients in the first row
    c_star[0] = c[0] / b[0];
    d_star[0] = d[0] / b[0];

    // update other coefficients iteratively
    for (size_t i = 1; i < N; ++i) {
        const double w = 1.0 / (b[i] - a[i] * c_star[i-1]);
        c_star[i] = c[i] * w;
        d_star[i] = (d[i] - a[i] * d_star[i-1]) * w;
    }

    // store the solution in the storage
    for (size_t i = N - 1; i-- > 0; ) {
        storage[i] = d_star[i] - c_star[i] * d[i+1];
    }
}

static void pprint_tridiag_matrix(const tridiagonal_mx_extended & mx,
std::ostream & out) {

```

```

const size_t diag_length = mx[0].size();
if (diag_length == 0) throw std::runtime_error("zero-length matrix");

auto add_spaces = [&out](const size_t n) {
    for (size_t tabs = 0; tabs < n; ++tabs) {
        out << " ";
    }
};

out << "0-" << mx[3][0] << ":\t\t" << mx[1][0] << ' ' << mx[2][0] <<
'\n';
for (size_t i = 1; i < diag_length - 1; ++i) {
    out << i << '-' << mx[3][i] << ":\t\t";
    add_spaces(i);
    out << mx[0][i] << ' ' << mx[1][i] << ' ' << mx[2][i] << '\n';
}
out << diag_length - 1 << '-' << mx[3][diag_length - 1] << ":\t\t";
add_spaces(diag_length - 1);
out << mx[0][diag_length - 1] << ' ' << mx[1][diag_length - 1] << '\n';

}

static void pprint_solution_row(const diagonal & d, std::ostream & os) {
    os << "solution: ";
    for (const auto & e: d)
        os << e << ' ';
    os << '\n';
}

Problem::Problem(model::IModel & model, const size_t n_iters):
    n_iters(n_iters),
    m(model),
    solver_x(model.x_dim()),
    solver_y(model.y_dim()),
    mx_x({
        // SLE contains 3 diagonals (a, b & c) and the RHS (d)
        // see https://quantstart.com/articles/Tridiagonal-Matrix-Solver-via-Thomas-Algorithm/
        diagonal(model.x_dim(), 0),
        diagonal(model.x_dim(), 0),
        diagonal(model.x_dim(), 0),
        diagonal(model.x_dim(), 0)
    }),
    mx_y({
        diagonal(model.y_dim(), 0),
        diagonal(model.y_dim(), 0),
        diagonal(model.y_dim(), 0),
        diagonal(model.y_dim(), 0)
    }),
    f_x(model.x_dim()),
    f_y(model.y_dim()) {}

```

```

void Problem::step() {
    if (current_step++ == n_iters) throw std::runtime_error("out of
iterations");
    // performs simulation step and stores the
    // result in the grid of the model
    const size_t x_dim = m.x_dim(), y_dim = m.y_dim();

    // first, solve the 1D subproblems in the horizontal direction
    // --> y_dim systems for each grid row
    // and update the current values at each node
    for (size_t y = 0; y < y_dim; ++y) {
        // solve SLE for row y
        // boundary conditions on edge:
        model::boundary_coefs bc = m.get_x_first_coefs(y);
        // unpack the duple into diagonals
        // once again, see https://quantstart.com/articles/Tridiagonal-
Matrix-Solver-via-Thomas-Algorithm/
        mx_x[1][0] = bc[0];
        mx_x[2][0] = bc[1];
        mx_x[3][0] = m.get_RHS_coefs_x(0, y);

        for (size_t x = 1; x < x_dim - 1; ++x) {
            // unpack triples into diagonals + right-hand side into d
            const model::tridiag_coefs tc = m.get_x_coefs(x, y);
            mx_x[0][x] = tc[0];
            mx_x[1][x] = tc[1];
            mx_x[2][x] = tc[2];
            mx_x[3][x] = m.get_RHS_coefs_x(x, y);
        }

        bc = m.get_x_last_coefs(y);
        mx_x[0][x_dim - 1] = bc[0];
        mx_x[1][x_dim - 1] = bc[1];
        mx_x[3][x_dim - 1] = m.get_RHS_coefs_x(x_dim - 1, y);

        // call solver and update current values in the row
        // std::cout << "matrix " << y << "\n";
        solver_x.solve(mx_x, f_x);
        if (VERBOSE) {
            pprint_tridiag_matrix(mx_x, std::cout);
            pprint_solution_row(f_x, std::cout);
        }
        // std::getchar();
        update_grid_row(y);
    }

    // then solve in the vertical direction -->
    // x_dim systems for each grid column
    // and update the current values at each node
    for (size_t x = 0; x < x_dim; ++x) {

```

```

        // solve SLE for column x
        // boundary conditions on edge:
        model::boundary_coefs bc = m.get_y_first_coefs(x);
        // unpack the duple into diagonals
        // once again, see https://quantstart.com/articles/Tridiagonal-
Matrix-Solver-via-Thomas-Algorithm/
        mx_y[1][0] = bc[0];
        mx_y[2][0] = bc[1];
        mx_y[3][0] = m.get_RHS_coefs_y(x, 0);

        for (size_t y = 1; y < y_dim - 1; ++y) {
            const model::tridiag_coefs tc = m.get_y_coefs(x, y);
            mx_y[0][y] = tc[0];
            mx_y[1][y] = tc[1];
            mx_y[2][y] = tc[2];
            mx_y[3][y] = m.get_RHS_coefs_y(x, y);
        }

        bc = m.get_y_last_coefs(x);
        mx_y[0][y_dim - 1] = bc[0];
        mx_y[1][y_dim - 1] = bc[1];
        mx_y[3][y_dim - 1] = m.get_RHS_coefs_y(x, y_dim - 1);

        // call solver and update current values in the column
        // std::cout << "matrix " << x << "\n";
        // std::getchar();
        solver_y.solve(mx_y, f_y);
        if (VERBOSE) {
            pprint_tridiag_matrix(mx_y, std::cout);
            pprint_solution_row(f_y, std::cout);
        }
        update_grid_col(x);
    }
}

void Problem::update_grid_row(const size_t y) {
    size_t i = 0;
    std::for_each(
        f_x.cbegin(), f_x.cend(),
        [&](const double & f) { m.set_current_value(i++, y, f); }
    );
}

void Problem::update_grid_col(const size_t x) {
    size_t i = 0;
    std::for_each(
        f_y.cbegin(), f_y.cend(),
        [&](const double & f) { m.set_current_value(x, i++, f); }
    );
}
}

```

plotter.hpp

```
#pragma once

#include <cctype>
#include <sstream>

#include "shared.hpp"

namespace plt {
    class GNUPlotWriter {
    public:
        GNUPlotWriter(const std::string & config);
        ~GNUPlotWriter();
        std::ostream & reciever() { return payload_buffer; }
        void flush_buffer();
        constexpr static std::string_view basic_gif_config =
            "set terminal gif size 800 800 animate delay 2 enhanced font"
            "'Verdana, 14'\n"
            "set output 'map.gif'\n"
            "set title 'Heat equation solution using TDMA solver'\n"
            "set xlabel 'X'\n"
            "set ylabel 'Y'\n"
            // "set xrange [0:199]\n"
            // "set yrange [0:99]\n"
            "set view map scale 1\n"
            "set palette color\n"
            "set pm3d map\n";
    private:
        FILE * pipe = nullptr;
        std::stringstream payload_buffer;
        // explicitly prohibit writing anything to the terminal
        constexpr static std::string_view command = "gnuplot 2> /dev/null";
    private:
        void throw_on_bad_pipe();
    };
}
```

plotter.cpp

```
#include "plotter.hpp"

namespace plt {
    GNUPlotWriter::GNUPlotWriter(const std::string & config) {
        pipe = popen(command.data(), "w");
        if (pipe == nullptr) throw std::runtime_error("failed to run GNUpot");
        fputs(config.c_str(), pipe);
    }
}
```

```
}

GNUPlotWriter::~GNUPlotWriter() {
    pclose(pipe);
}

void GNUPlotWriter::flush_buffer() {
    throw_on_bad_pipe();
    fputs("splot '-' matrix with image\n", pipe);
    fputs(payload_buffer.str().c_str(), pipe);
    fputs("e\n", pipe);
    std::stringstream().swap(payload_buffer);
}

void GNUPlotWriter::throw_on_bad_pipe() {
    if (pipe == nullptr)
        throw std::runtime_error("failed to open GNUPlot subprocess");
}
}
```