SPRAWOZDANIE		Data 14.01.2025
Temat:	Metoda Elementów Skończonych	
Autor:	Paweł Socała	

1. Wstęp

Sprawozdanie przedstawia implementację Metody Elementów Skończonych w kontekście symulacji ustalonych procesów cieplnych. Oprogramowanie zostało napisane przy pomocy języka *C++*.

2. Analiza programu

Na początku opisu podano strukturę projektu w *Visual Studio Code*, aby przedstawienie oprogramowania było bardziej czytelne.

```
GRID/
├── include/                        # Pliki nagłówkowe projektu
│   ├── Element.h
│   ├── Node.h
│   ├── GlobalData.h
│   ├── Grid.h
│   ├── Integration.h
│   └── FEMSolver.h
├── src/                            # Pliki źródłowe
│   ├── Element.cpp
│   ├── Node.cpp
│   ├── GlobalData.cpp
│   ├── Grid.cpp
│   ├── Integration.cpp
│   ├── FEMSolver.cpp
│   └── main.c
├── data/                           # Dane użyte do symulacji
│   ├── data.txt
│   └── xy_nodes.txt
├── results/                        # Wyniki symulacji
│   ├── global_C_matrix.txt
│   ├── global_P_matrix.txt
│   ├── global_t_matrix.txt
│   ├── global_Hbc_matrix.txt
│   └── simulation_temperatures.txt
└── simulation.exe                  # Plik wykonywalny
```

Działanie programu krok po kroku:

1. Klasa **Node**

```
class Node {
private:
    double x, y;
    int BC;

public:
    Node();
    Node(double p_x, double p_y, int BC);
    double get_x() const;
    double get_y() const;
    int get_BC() const;
    void display_node() const;
};
```

Klasa **Node** służy do reprezentacji pojedynczego węzła siatki. Zawiera pola: x, y – współrzędne węzła oraz BC – warunek brzegowy (kontakt z otoczeniem). Klasa posiada również szereg metod służących do zwracania oraz wyświetlania danych węzła.

2. Klasa **Element**

```
class Element {
private:
    Node nodes_xy[4];
    vector<int> ID;
    vector<vector<double>> Hbc;
    vector<double> P;

public:
    Element(const Node& n1, const Node& n2, const Node& n3, const Node& n4);
    void set_ID(int node_1, int node_2, int node_3, int node_4);
    void display_ID() const;
    const Node* get_nodes() const;
    const vector<int>& get_ID() const;
    void set_Hbc(const vector<vector<double>>& hbc);
    const vector<vector<double>>& get_Hbc() const;
    void set_P(const vector<double>& p);
    const vector<double>& get_P() const;
};
```

Klasa **Element** służy do reprezentacji pojedynczego elemntu siatki, który składa się z 4 węzłów (Node). Klasa przechowuje także dodatkowe informacje takie jak ID elementu, lokalną macierz Hbc (reprezentuje przepływ ciepła na krawędzi elementu) oraz wektor obciążeń P.

3. Pobranie danych do symulacji – Klasa **GlobalData**

```
void GlobalData::read_file() {
    try {
        ifstream data_file("../Grid/data/data.txt");
        if (!data_file.is_open()) {
            throw runtime_error("File: data.txt not found.");
```

```
        }

        if (!(data_file >> simulation_time >> simulation_step_time >> conductivity >> alfa
>> ambient_temp >> initial_temp >> density >> specific_heat >> nN >> nE >> nH >> nW >> H >>
W)) {
            throw runtime_error("Error reading data from file: data.txt");
        }

        if (simulation_time < 0) {
            throw runtime_error("Simulation time cannot be negative.");
        }

        if (simulation_step_time <= 0 || conductivity <= 0 || alfa <= 0 || density <= 0 ||
specific_heat <= 0
                || nN <= 0 || nE <= 0 || nH <= 0 || nW <= 0 || H <= 0 || W <= 0) {
            throw runtime_error("Simulation data must be positive.");
        }

        data_file.close();

        ifstream xy_nodes_file("../Grid/data/xy_nodes.txt");
        if (!xy_nodes_file.is_open()) {
            throw runtime_error("File: xy_nodes.txt not found.");
        }

        string line;
        while (getline(xy_nodes_file, line)) {
            double x, y;
            int BC;
            if (!(istringstream(line) >> x >> y >> BC)) {
                cout << x << y << BC;
                throw runtime_error("Failed to read line: " + line);
            }
            nodes_xy.emplace_back(x, y, BC);
        }
        xy_nodes_file.close();

        if (nodes_xy.empty()) {
            cerr << "Error: No nodes were loaded from xy_nodes.txt" << endl;
        }

    } catch (const runtime_error& e) {
        cerr << "Error: " << e.what() << endl;
    }
}
```

Funkcja read_file() jest metodą klasy GlobalData, która wczytuje dane z dwóch plików: data.txt (dane symulacji) i xy_nodes.txt (współrzędne węzłów). Funkcja posiada obsługę wyjątków, która zapobiega błędom podczas wczytywania danych oraz sprawdza ich poprawność. Funkcja wczytuje dane do zmiennych: simulation_time, simulation_step_time, conductivity, alfa, ambient_temp, initial_temp, density, specific_heat, nN , nE , nH oraz nW. Następnie odczytuje dane węzłów (x, y - współrzędne węzłów oraz BC - wartość warunku brzegowego sprawdzającego kontakt z otoczeniem). Po odczytaniu współrzędnych funkcja tworzy obiekty klasy Node.

4. Klasa **Grid**

```
class Grid {
```

```
private:
    double nN, nE, nW, nH, height, width;
    vector<Node> nodes;
    static vector<Node> nodes_xy;
    vector<Element> elements;

public:
    Grid();
    Grid(double p_nN, double p_nE, double p_nW, double p_nH, double p_height, double
p_width);
    vector<Element>& get_elements();
    void create_elements();
    void display_grid_data();
};
```

Klasa **Grid** reprezentuje siatkę elementów używaną w metodzie elementów skończonych. Siatka składa się z węzłów (Node) oraz elementów (Element), które tworzą strukturę geometryczną symulacji. Klasa zarządza generowaniem elementów siatki oraz ich powiązaniem z węzłami.

```
Grid::Grid(double p_nN, double p_nE, double p_nW, double p_nH, double p_height, double
p_width) : nN(p_nN), nE(p_nE), nW(p_nW), nH(p_nH), height(p_height), width(p_width) {
    create_elements();
}
```

Konstruktor klasy Grid inicjalizuje siatkę na podstawie wczytanych parametrów oraz automatycznie uruchamia metodę generującą elementy siatki

```
void Grid::create_elements() {
    nodes_xy = GlobalData::get_nodes();
    size_t nodes_num = nodes_xy.size();

    if (nodes_num != nN) {
        cerr << "Wrong number of nodes." << endl;
        return;
    }

    int nodes_per_row = sqrt(nN);
    int nodes_per_col = nN / nodes_per_row;

    for (int i = 0; i < nodes_per_col - 1; ++i) {
        for (int j = 0; j < nodes_per_row - 1; ++j) {
            int node_1 = i * nodes_per_row + j;
            int node_2 = node_1 + 1;
            int node_3 = node_1 + nodes_per_row + 1;
            int node_4 = node_1 + nodes_per_row;

            Element new_element(nodes_xy[node_1], nodes_xy[node_2], nodes_xy[node_3],
nodes_xy[node_4]);
            new_element.set_ID(node_1, node_2, node_3, node_4);

            elements.push_back(new_element);
        }
    }
}
```

Metoda create_elements() tworzy elementy na podstawie obiektów klasy Node. Węzły w każdym elemencie są numerowane zgodnie z ruchem wskazówek zegara, zaczynając od węzła w lewym górnym rogu. Na końcu funkcja przypisuje ID każdemu z elementów.

5. Klasa **Integration**

```
class Integration {
private:
    static const double x2[2];
    static const double w2[2];
    static const double x3[3];
    static const double w3[3];
    static const double x4[4];
    static const double w4[4];
    static const double x9[9];
    static const double w9[9];
    static const double x16[16];
    static const double w16[16];
public:
    Integration();
    double gauss_integration_2D(function<double(double, double)> f, int n, double a, double
b, double c, double d); // funkcja | liczba punktów | granice całkowania
    void display_results(double result);
    vector<double> get_weights(int order);
    vector<double> get_points(int order);
};
```

Klasa **Integration** to klasa służąca do całkowania metodą kwadratury Gaussa. Jej szczegółowe zastosowanie określono w opisie kolejnej klasy. Kasa przechowuje tablice punktów i wag Gaussa dla różnych rzędów (stopni kwadratury). Punkty i wagi są ustalonymi wartościami i odpowiadają optymalnym pozycjom i wartościom, które minimalizują błąd numeryczny podczas całkowania.

```
double Integration::gauss_integration_2D(function<double(double, double)> f, int n, double
a, double b, double c, double d) {
    double suma = 0.0;
    const double* x;
    const double* w;

    switch (n) {
        case 2:
            x = x2;
            w = w2;
            break;
        case 3:
            x = x3;
            w = w3;
            break;
        case 4:
            x = x4;
            w = w4;
            break;
        case 9:
            x = x9;
            w = w9;
            break;
        case 16:
```

```
            x = x16;
            w = w16;
            break;
        default:
            cerr << "Wrong number of points." << endl;
            return 0;
    }

    double c1 = (b + a) / 2.0;
    double d1 = (b - a) / 2.0;
    double c2 = (d + c) / 2.0;
    double d2 = (d - c) / 2.0;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            suma += w[i] * w[j] * f(d1 * x[i] + c1, d2 * x[j] + c2);
        }
    }

    return d1 * d2 * suma;
}
```

Funkcja wykonuje numeryczne całkowanie dwuwymiarowe (2D) metodą kwadratury Gaussa. **Parametry:**

- **f**: Funkcja, która ma być całkowana, przyjmująca dwa argumenty: $f(x, y)$
- **n**: Liczba punktów całkowania (rząd kwadratury: 2, 3, 4, 9 lub 16).
- **a, b**: Dolna i górna granica całkowania w osi x.
- **c, d**: Dolna i górna granica całkowania w osi y.

Po przyjęciu odpowiedniej ilości punktów całkowania metoda mapuje punktów z układu [-1, 1] na rzeczywisty. Najpierw przygotowywane są odpowiednie dane:

- $\boldsymbol{c_1}$ Środek przedziału [a,b] w osi x.
- $\boldsymbol{d_1}$ Połowa długości przedziału [a,b] w osi x.
- $\boldsymbol{c_2}$ Środek przedziału [c,d] w osi y.
- $\boldsymbol{c_2}$ Połowa długości przedziału [c,d] w osi y.

Pętla podwójna przetwarza punkty kwadratury x[i] (dla osi x) i x[j] (dla osi y) dla każdej pary punktów. Kolejno funkcja $f(x, y)$ obliczana jest w przekształconych współrzędnych rzeczywistych: f(d1 * x[i] + c1, d2 * x[j] + c2). Każdy wynik funkcji fff jest mnożony przez odpowiadające wagi w[i] i w[j] oraz dodawany do sumy, która określa przybliżoną wartość całki dwuwymiarowej.

6. Klasa **FEMSolver**

```
class FEMSolver {
private:
```

```
    Grid& grid;
    vector<vector<double>> local_H_matrices, local_C_matrices;
    vector<double> P_global;
public:
    explicit FEMSolver(Grid& grid, double alpha, double ambient_temperature);
    void display_matrix(vector<vector<double>>& matrix);
    void compute_jacobian(const Element& element, double xi, double eta, double J[2][2])
const;
    double compute_jacobian_determinant(double J[2][2]) const;
    void compute_inverse_jacobian(double J[2][2], double invJ[2][2]) const;
    void calculate_Hbc_matrix(double conductivity);
    double calculate_H_integrand(const Element& element, double conductivity, int i, int j,
double xi, double eta) const;
    void aggregate_Hbc_matrix(vector<vector<double>>& H_global, int nodes_num) const;
    void calculate_local_Hbc_matrix(double alpha);
    void integrate_Hbc_on_edge(const Node& node1, const Node& node2, double alpha,
vector<vector<double>>& Hbc) const;
    void compute_edge_jacobian(const Node& node1, const Node& node2, double xi, double&
detJ) const;
    void calculate_P_vector(double alpha, double ambient_temperature);
    void aggregate_P_vector(vector<double>& P_global, int nodes_num) const;
    void solve_system(vector<vector<double>>& H_global, vector<double>& P_global,
vector<double>& t_global);
    void calculate_C_matrix(double density, double specific_heat);
    void aggregate_C_matrix(vector<vector<double>>& C_global, int nodes_num) const;
    void simulate_time(vector<vector<double>>& H_global, vector<vector<double>>& C_global,
vector<double>& P_global, vector<double>& t_initial, double time_step, double total_time);
};
```

Klasa **FEMSolver** jest odpowiedzialna za rozwiązanie układów równań z zakresu metody elementów skończonych dla problemu przewodzenia ciepła. Jej zadaniem jest przeprowadzenie wszystkich obliczeń niezbędnych do rozwiązania układu równań i obliczeń związanych z symulacją w czasie, uwzględniając zarówno macierze przewodzenia ciepła, macierze pojemności cieplnej, jak i wektory źródeł ciepła.

```
FEMSolver::FEMSolver(Grid& grid, double alpha, double ambient_temperature) : grid(grid) {
    local_H_matrices.resize(grid.get_elements().size(), vector<double>(4, 0.0));
    calculate_local_Hbc_matrix(alpha);
}
```

Konstruktor klasy inicjalizuje siatkę grid oraz wartości współczynnika przewodzenia ciepła (alpha) i temperaturę otoczenia. Wywołuje on również metodę calculate_local_Hbc_matrix(), którą została przedstawiona w kolejnym etapie opisu.

Macierze J ----------

```
void FEMSolver::compute_jacobian(const Element& element, double xi, double eta, double
J[2][2]) const {
    double dN_dxi[4] = {
        -0.25 * (1 - eta),
         0.25 * (1 - eta),
         0.25 * (1 + eta),
        -0.25 * (1 + eta)
    };
```

```
    double dN_deta[4] = {
        -0.25 * (1 - xi),
        -0.25 * (1 + xi),
         0.25 * (1 + xi),
         0.25 * (1 - xi)
    };

    const Node* nodes = element.get_nodes();
    fill(&J[0][0], &J[0][0] + 4, 0.0);

    for (int i = 0; i < 4; ++i) {

        J[0][0] += dN_dxi[i] * nodes[i].get_x();
        J[0][1] += dN_dxi[i] * nodes[i].get_y();
        J[1][0] += dN_deta[i] * nodes[i].get_x();
        J[1][1] += dN_deta[i] * nodes[i].get_y();
    }
}
```

Metoda ta oblicza macierz Jacobiego J dla danego elementu w określonym punkcie w układzie współrzędnych naturalnych (używając współrzędnych lokalnych: ξ i η). Macierz Jacobiego pozwala na obliczenie odwrotnej macierzy J co pozwala przekształcać pochodne funkcji interpolacyjnych z układu lokalnego do globalnego układu współrzędnych.

Metoda ta zaczyna od obliczenia pochodnych funkcji interpolacyjnych względem współrzędnych naturalnych ξ i η. Następnie, przy użyciu współrzędnych węzłów elementu (element.get_nodes()), funkcja oblicza elementy macierzy Jacobiego J. Każdy element macierzy jest sumą iloczynu odpowiednich funkcji interpolacyjnych i współrzędnych x lub y.

```
double FEMSolver::compute_jacobian_determinant(double J[2][2]) const {
    return J[0][0] * J[1][1] - J[0][1] * J[1][0];
}
```

Oblicza wyznacznik w klasyczny sposób dla macierzy 2 X 2.

```
void FEMSolver::compute_inverse_jacobian(double J[2][2], double invJ[2][2]) const {
    double detJ = compute_jacobian_determinant(J);

    if (abs(detJ) < 1e-12) {
        cerr << "Error: Jacobian determinant is close to 0! Skipping element." << endl;
        return;
    }

    invJ[0][0] = J[1][1] / detJ;
    invJ[0][1] = -J[0][1] / detJ;
    invJ[1][0] = -J[1][0] / detJ;
    invJ[1][1] = J[0][0] / detJ;
}
```

Metoda ta oblicza odwrotność macierzy Jacobiego J. Funkcja najpierw oblicza wyznacznik macierzy Jacobiego za pomocą funkcji compute_jacobian_determinant().

Jeśli wyznacznik jest różny od zera, funkcja oblicza odwrotność macierzy Jacobiego zgodnie ze wzorem dla macierzy 2x2.

Macierze H ----------

```
double FEMSolver::calculate_H_integrand(const Element& element, double conductivity, int i,
int j, double xi, double eta) const {
    double dN_dxi[4] = {
        -0.25 * (1 - eta),
         0.25 * (1 - eta),
         0.25 * (1 + eta),
        -0.25 * (1 + eta)
    };

    double dN_deta[4] = {
        -0.25 * (1 - xi),
        -0.25 * (1 + xi),
         0.25 * (1 + xi),
         0.25 * (1 - xi)
    };

    double J[2][2];
    compute_jacobian(element, xi, eta, J);
    double detJ = compute_jacobian_determinant(J);

    double invJ[2][2];
    compute_inverse_jacobian(J, invJ);

    const auto& nodes = element.get_nodes();

    double dN_dx[4], dN_dy[4];
    for (int k = 0; k < 4; ++k) {
        dN_dx[k] = invJ[0][0] * dN_dxi[k] + invJ[0][1] * dN_deta[k];
        dN_dy[k] = invJ[1][0] * dN_dxi[k] + invJ[1][1] * dN_deta[k];
    }

    return conductivity * (dN_dx[i] * dN_dx[j] + dN_dy[i] * dN_dy[j]) * detJ;
}
```

Metoda ta oblicza składnik całki, który jest używany do obliczenia elementów globalnej macierzy przewodnictwa. Szczegóły działania:

Obliczenie pochodnych funkcji kształtu względem współrzędnych lokalnych (xi, eta):

dN_dxi i dN_deta to pochodne funkcji kształtu (N) względem współrzędnych lokalnych (xi i eta) w układzie współrzędnych elementu skończonego. Są to funkcje w układzie lokalnym, które zależą od geometrycznych właściwości elementu (czterech węzłów w tym przypadku).

Obliczenie macierzy Jacobiego i jej wyznacznika:

compute_jacobian() oblicza macierz Jacobiego, która jest używana do przekształcania pochodnych w układzie lokalnym na układ globalny.

compute_jacobian_determinant() oblicza wyznacznik macierzy Jacobiego, który jest potrzebny do obliczenia objętości lub powierzchni elementu w przestrzeni globalnej.

compute_inverse_jacobian() oblicza odwrotność macierzy Jacobiego, co jest potrzebne do przekształcania pochodnych funkcji kształtu z układu lokalnego na globalny.

Obliczanie pochodnych funkcji kształtu w przestrzeni globalnej:

Zmienna dN_dx i dN_dy przechowują pochodne funkcji kształtu w kierunkach globalnych (czyli względem współrzędnych przestrzennych x i y). Są one obliczane poprzez przekształcenie lokalnych pochodnych (dN_dxi, dN_deta) z wykorzystaniem odwrotności macierzy Jacobiego.

Zwrócenie składnika całki:

Na końcu funkcja zwraca składnik całki, który jest wynikiem mnożenia przewodności conductivity przez kwadrat pochodnych funkcji kształtu w przestrzeni globalnej, a także przez wyznacznik macierzy Jacobiego (detJ).

```
void FEMSolver::calculate_Hbc_matrix(double conductivity) {
    Integration integrator;
    local_H_matrices.clear();

    for (const auto& element : grid.get_elements()) {
        vector<double> H(16, 0.0);

        const auto& nodes = element.get_nodes();
        const auto& ID = element.get_ID();

        cout << "Element ID: ";
        for (size_t i = 0; i < ID.size(); ++i) {
            cout << ID[i] << " ";
        }
        cout << endl;

        for (int i = 0; i < 4; ++i) {
            for (int j = 0; j < 4; ++j) {
                H[i * 4 + j] = integrator.gauss_integration_2D(
                    [this, &element, conductivity, i, j](double xi, double eta) -> double {
                        return this->calculate_H_integrand(element, conductivity, i, j, xi,
eta);
                    }, 16, -1, 1, -1, 1);
            }
        }
        const auto& Hbc_local = element.get_Hbc();
        for (int i = 0; i < 4; ++i) {
            for (int j = 0; j < 4; ++j) {
                H[i * 4 + j] += Hbc_local[i][j];
            }
        }

        local_H_matrices.push_back(H);
    }
}
```

Celem tej funkcji jest obliczenie lokalnej macierzy H dla każdego elementu w siatce, uwzględniając warunki brzegowe Hbc. Wartości macierzy H są obliczane za pomocą całkowania numerycznego (przy użyciu całkowania Gaussa w 2D).

Pętle te iterują po wszystkich parach indeksów (i, j) odpowiadających elementom macierzy H. Wywołana jest funkcja gauss_integration_2D z obiektu integrator, która służy do numerycznego obliczania całek w dwóch wymiarach (2D). Pierwszym argumentem przekazywanym do funkcji całkującej jest lambda, która reprezentuje funkcję do całkowania. Jest to funkcja anonimowa, która jest przekazywana do funkcji całkowania. Lambda ta wykonuje obliczenia dla każdego punktu całkowania w przestrzeni lokalnych współrzędnych (xi, eta). Dla każdego punktu całkowania (xi, eta) obliczana jest wartość funkcji całkowanej za pomocą metody calculate_H_integrand(). Argumenty tej funkcji lambda:

- **xi** i **eta** to współrzędne lokalne, które reprezentują punkty w jednostkowym obszarze elementu, używane w całkowaniu numerycznym.

- **element** to aktualny element siatki, którego macierz H jest obliczana.

- **conductivity** to współczynnik przewodności materiału, który wpływa na wartość macierzy H.

- **i, j** to indeksy wiersza i kolumny macierzy H, które są obliczane.

Po wykonaniu całkowania, uzyskiwana jest wartość, która jest przypisana do odpowiedniej pozycji w macierzy H: H[i * 4 + j]. Mnożenie przez 4 jest konieczne do odpowiedniego indeksowania macierzy jednowymiarowej, która przechowuje dane dla 4x4 macierzy. Na końcu dodajemy do macierzy wartości związane z warunkami brzegowymi Hbc (obliczanie lokalnych macierzy Hbc pokazano w kolejnym etapie opisu), które modyfikują obliczoną wcześniej macierz H.

```
void FEMSolver::aggregate_Hbc_matrix(vector<vector<double>>& H_global, int nodes_num) const
{
    H_global.assign(nodes_num, vector<double>(nodes_num, 0.0));
    const auto& elements = grid.get_elements();

    for (size_t elem_idx = 0; elem_idx < elements.size(); ++elem_idx) {
        const auto& element = elements[elem_idx];
        const auto& H_local = local_H_matrices[elem_idx];
        const auto& ID = element.get_ID();

        for (int i = 0; i < 4; ++i) {
            for (int j = 0; j < 4; ++j) {
                if (ID[i] < nodes_num && ID[j] < nodes_num) {
                    H_global[ID[i]][ID[j]] += H_local[i * 4 + j];
                } else {
                    cerr << "Invalid global index: " << ID[i] << " or " << ID[j] << endl;
                }
            }
        }
    }
       // funkcje wypisujące macierze w terminalu oraz wpisujące do plików z wynikami
}
```

Metoda jest odpowiedzialna za zebranie (agregowanie) lokalnych macierzy Hbc dla wszystkich elementów w siatce do jednej globalnej macierzy H_global.

Głowna część kodu odpowiada za iterację po wszystkich parach węzłów dla elementu. ID[i] i ID[j] to globalne indeksy węzłów tego elementu. Jeśli oba indeksy są poprawne (mieszczą się w zakresie liczby węzłów globalnych nodes_num), to wartość lokalnej macierzy H_local[i * 4 + j] jest dodawana do globalnej macierzy H_global[ID[i]][ID[j]].

```
void FEMSolver::integrate_Hbc_on_edge(const Node& node1, const Node& node2, double alpha,
vector<vector<double>>& Hbc) const {
    Integration integrator;
    vector<double> xi_points = integrator.get_points(4);
    vector<double> weights = integrator.get_weights(4);

    double L = sqrt(pow(node2.get_x() - node1.get_x(), 2) + pow(node2.get_y() -
node1.get_y(), 2));

    for (int k = 0; k < xi_points.size(); ++k) {
        double xi = xi_points[k];
        double weight = weights[k];
        double N1 = 0.5 * (1 - xi);
        double N2 = 0.5 * (1 + xi);
        double detJ = L / 2;
        double Hbc_value = alpha * weight * detJ;

        Hbc[0][0] += Hbc_value * N1 * N1;
        Hbc[0][1] += Hbc_value * N1 * N2;
        Hbc[1][0] += Hbc_value * N2 * N1;
        Hbc[1][1] += Hbc_value * N2 * N2;
    }
}
```

Funkcja oblicza wkład z warunków brzegowych (oznaczony jako Hbc) na krawędzi elementu, przeprowadzając całkowanie numeryczne za pomocą metody Gaussa (w tym przypadku dla jednej krawędzi). Obliczana jest długość krawędzi L między węzłami node1 i node2 w przestrzeni 2D, używając wzoru na odległość euklidesową.
Następnie w pętli (iterującej przez punkty Gaussa) dla każdego punktu ξ, obliczane są funkcje kształtu N1 i N2 (które są funkcjami liniowymi zależnymi od ξ) w celu ujęcia kształtu elementu na tej krawędzi. Na podstawie tych funkcji kształtu oraz wag oblicza się wkład Hbc na tej krawędzi, który na końcu dodawany jest do macierzy Hbc.

```
void FEMSolver::calculate_local_Hbc_matrix(double alpha) {
    for (auto& element : grid.get_elements()) {
        vector<vector<double>> Hbc_local(4, vector<double>(4, 0.0));

        const Node* nodes = element.get_nodes();
        for (int edge = 0; edge < 4; ++edge) {
            const Node& node1 = nodes[edge];
            const Node& node2 = nodes[(edge + 1) % 4];

            if (node1.get_BC() && node2.get_BC()) {
                vector<vector<double>> Hbc_edge(2, vector<double>(2, 0.0));
                integrate_Hbc_on_edge(node1, node2, alpha, Hbc_edge);
```

```
                int local_indices[2] = {edge, (edge + 1) % 4};

                for (int i = 0; i < 2; ++i) {
                    for (int j = 0; j < 2; ++j) {
                        Hbc_local[local_indices[i]][local_indices[j]] += Hbc_edge[i][j];
                    }
                }
            }
        }

        element.set_Hbc(Hbc_local);

        //funkcja wyświetlająca macierz
    }
}
```

Funkcja ta oblicza lokalną macierz Hbc dla każdego elementu, uwzględniając warunki brzegowe na wszystkich czterech krawędziach elementu. a każdej krawędzi sprawdzane jest, czy oba węzły na tej krawędzi mają przypisane warunki brzegowe (sprawdzane przez node1.get_BC() oraz node2.get_BC()). Jeśli oba węzły krawędzi mają przypisane warunki brzegowe, to wywoływana jest funkcja integrate_Hbc_on_edge(), aby obliczyć wkład dla tej krawędzi. Po obliczeniu wkładów dla wszystkich krawędzi, lokalna macierz Hbc jest przypisywana do elementu za pomocą element.set_Hbc(Hbc_local). Set_Hbc jest później wykorzystywane do dodania wkładu do macierzy H.

Wektory P ----------

```
void FEMSolver::calculate_P_vector(double alpha, double ambient_temperature) {
    cout << "-----------------------------------" << endl;
    cout << "Local P vectors for elements:" << endl << endl;

    for (auto& element : grid.get_elements()) {
        vector<double> P_local(4, 0.0);
        const Node* nodes = element.get_nodes();

        for (int edge = 0; edge < 4; ++edge) {
            const Node& node1 = nodes[edge];
            const Node& node2 = nodes[(edge + 1) % 4];

            if (node1.get_BC() && node2.get_BC()) {
                Integration integrator;
                vector<double> xi_points = integrator.get_points(2);
                vector<double> weights = integrator.get_weights(2);
                double L = sqrt(pow(node2.get_x() - node1.get_x(), 2) + pow(node2.get_y() -
node1.get_y(), 2));

                for (int k = 0; k < xi_points.size(); ++k) {
                    double xi = xi_points[k];
                    double weight = weights[k];
                    double N1 = 0.5 * (1 - xi);
                    double N2 = 0.5 * (1 + xi);
                    double detJ = 0.5 * L;
                    double q = alpha * ambient_temperature * weight * detJ;

                    P_local[edge] += q * N1;
                    P_local[(edge + 1) % 4] += q * N2;
                }
```

```
            }
        }

        element.set_P(P_local);

        for (const auto& value : P_local) {
            cout << value << " ";
        }
        cout << endl << endl;
    }
}
```

Funkcja ta oblicza lokalny wektor obciążeń PPP dla każdego elementu w siatce. Wektor PPP reprezentuje obciążenia w postaci sił, które są generowane przez warunki brzegowe. Funkcja iteruje po wszystkich elementach w siatce. Dla każdego elementu tworzony jest lokalny wektor obciążeń. Dla każdej krawędzi sprawdzane jest, czy oba węzły krawędzi mają przypisane warunki brzegowe. Jeśli oba węzły mają warunki brzegowe, funkcja oblicza wkład do wektora obciążeń P z tej krawędzi. Używa do tego metody całkowania Gaussa. Oblicza również długość krawędzi L, a następnie wkład do wektora obciążeń P, przy czym każdy wkład jest pomnożony przez odpowiednią wagę, współczynnik alpha, temperaturę otoczenia i długość krawędzi.

```
void FEMSolver::aggregate_P_vector(vector<double>& P_global, int nodes_num) const {
    P_global.assign(nodes_num, 0.0);
    const auto& elements = grid.get_elements();

    for (const auto& element : elements) {
        const auto& P_local = element.get_P();
        const auto& ID = element.get_ID();

        for (int i = 0; i < 4; ++i) {
            if (ID[i] < nodes_num) {
                P_global[ID[i]] += P_local[i];
            } else {
                cerr << "Invalid global index: " << ID[i] << endl;
            }
        }
    }
       // Wyświetlenie wektora oraz wczytanie do pliku
}
```

Funkcja ta agreguje lokalne wektory obciążeń obliczone dla każdego elementu do globalnego wektora obciążeń.

Macierze C ----------

```
void FEMSolver::calculate_C_matrix(double density, double specific_heat) {
    Integration integrator;
    local_C_matrices.clear();

    int order = 4;
    vector<double> gauss_points = integrator.get_points(order);
    vector<double> gauss_weights = integrator.get_weights(order);
```

```
    for (const auto& element : grid.get_elements()) {
        vector<double> C_local(16, 0.0);

        const auto& nodes = element.get_nodes();

        for (int i = 0; i < order; ++i) {
            for (int j = 0; j < order; ++j) {
                double xi = gauss_points[i];
                double eta = gauss_points[j];
                double weight = gauss_weights[i] * gauss_weights[j];

                double N[4] = {
                    0.25 * (1 - xi) * (1 - eta),
                    0.25 * (1 + xi) * (1 - eta),
                    0.25 * (1 + xi) * (1 + eta),
                    0.25 * (1 - xi) * (1 + eta)
                };

                double J[2][2];
                compute_jacobian(element, xi, eta, J);
                double detJ = compute_jacobian_determinant(J);

                for (int k = 0; k < 4; ++k) {
                    for (int l = 0; l < 4; ++l) {
                        C_local[k * 4 + l] += density * specific_heat * N[k] * N[l] * detJ
* weight;
                    }
                }
            }
        }

        local_C_matrices.push_back(C_local);
    }
}
```

Funkcja ta oblicza lokalne macierze pojemności cieplnej (C) dla każdego elementu w siatce. Macierz pojemności cieplnej jest podstawą do analizy termicznej, gdzie opisuje, jak elementy magazynują ciepło w procesie przewodzenia ciepła. W tej funkcji dla każdego elementu w siatce obliczane są wkłady do macierzy lokalnej C.
Funkcja iteruje przez wszystkie elementy w siatce. Dla każdego elementu (np. czworokąta) oblicza się lokalną macierz pojemności cieplnej. Dla każdego punktu całkowania na siatce, w funkcji calculate_C_matrix obliczane są wartości funkcji kształtu N1, N2, N3, N4 dla każdego węzła w elemencie. Kolejno obliczamy macierz Jacobiego i jej wyznacznik.
Lokalna macierz pojemności cieplnej jest obliczana poprzez sumowanie wkładów dla każdego punktu całkowania zgodnie ze wzorem:

```
C_local[k * 4 + l] += density * specific_heat * N[k] * N[l] * detJ * weight;
```

```
void FEMSolver::aggregate_C_matrix(vector<vector<double>>& C_global, int nodes_num) const {
    C_global.assign(nodes_num, vector<double>(nodes_num, 0.0));
    const auto& elements = grid.get_elements();

    for (size_t elem_idx = 0; elem_idx < elements.size(); ++elem_idx) {
        const auto& element = elements[elem_idx];
```

```
        const auto& C_local = local_C_matrices[elem_idx];
        const auto& ID = element.get_ID();

        for (int i = 0; i < 4; ++i) {
            for (int j = 0; j < 4; ++j) {
                if (ID[i] < nodes_num && ID[j] < nodes_num) {
                    C_global[ID[i]][ID[j]] += C_local[i * 4 + j];
                } else {
                    cerr << "Invalid global index: " << ID[i] << " or " << ID[j] << endl;
                }
            }
        }
    }
       // Wypisanie macierzy oraz wpisanie wyników do plików
}
```

Metoda jest odpowiedzialna za zebranie (agregowanie) lokalnych macierzy C_local dla wszystkich elementów w siatce do jednej globalnej macierzy C_global.

Temperatura ----------

```
void FEMSolver::simulate_time(vector<vector<double>>& H_global, vector<vector<double>>&
C_global, vector<double>& P_global, vector<double>& t_initial, double time_step, double
total_time) {
    int num_nodes = H_global.size();
    vector<double> t_current = t_initial;
    vector<double> t_next(num_nodes, 0.0);
    vector<vector<double>> A(num_nodes, vector<double>(num_nodes, 0.0));
    vector<double> b(num_nodes, 0.0);

    ofstream output_file("../Grid/results/simulation_temperatures.txt");
    if (!output_file.is_open()) {
        cerr << "Error opening file for writing results." << endl;
        return;
    }
    output_file << fixed << setprecision(5);
    output_file << "Simulation results:\n\n";

    for (double time = 50.0; time <= total_time; time += time_step) {
        output_file << "Time: " << time << " s\n";

        for (int i = 0; i < num_nodes; ++i) {
            for (int j = 0; j < num_nodes; ++j) {
                A[i][j] = C_global[i][j] / time_step + H_global[i][j];
            }
        }

        cout << "-----------------------------------" << endl;
        cout << "Matrix [H] + [C]/dT at time: " << time << " s" << endl;
        for (const auto& row : A) {
            for (const auto& value : row) {
                cout << fixed << setprecision(5) << value << " ";
            }
            cout << endl;
        }

        for (int i = 0; i < num_nodes; ++i) {
            b[i] = P_global[i];
            for (int j = 0; j < num_nodes; ++j) {
                b[i] += C_global[i][j] * t_current[j] / time_step;
            }
        }
```

```
        cout << "Vector P ([{P} + {[C]/dT}*{T0}]) at time: " << time << " s" << endl;
        for (const auto& value : b) {
            cout << fixed << setprecision(5) << value << " ";
        }
        cout << endl;

        solve_system(A, b, t_next);

    }
    output_file.close();
}
```

Metoda ta wykonuje symulację zmian temperatury w czasie, bazując na rozwiązaniu układu równań różniczkowych, które opisują dynamiczne przewodzenie ciepła w materiale. Funkcja jako parametry przyjmuje: macierze globalne H i C oraz wektor globalny P. Dla każdej iteracji czasowej, macierz A jest obliczana jako suma macierzy C_global / time_step oraz macierzy przewodzenia H_global. Ta macierz jest podstawą do rozwiązania układu równań.

Wektor b jest obliczany w oparciu o wartości wektora P_global oraz wkład od poprzednich temperatur (z wektora t_current) przeskalowanych przez macierz C_global i dzielonych przez krok czasowy.

Metoda wywołuje funkcję solve_system() która rozwiązuje układ równań:

$$\left([H]+\frac{[C]}{\Delta\tau}\right)\{t_1\}-\left(\frac{[C]}{\Delta\tau}\right)\{t_0\}+\{P\}=0$$

```
void FEMSolver::solve_system(vector<vector<double>>& H_global, vector<double>& P_global,
vector<double>& t_global) {
    int n = H_global.size();

    vector<vector<double>> A = H_global;
    vector<double> b = P_global;
    vector<double> x(n, 0.0);

    for (int i = 0; i < n; i++) {
        int max_row = i;
        for (int k = i + 1; k < n; k++) {
            if (abs(A[k][i]) > abs(A[max_row][i])) {
                max_row = k;
            }
        }

        if (abs(A[max_row][i]) < 1e-12) {
            cerr << ("The matrix is singular, the system cannot be solved.") << endl;
        }

        swap(A[i], A[max_row]);
        swap(b[i], b[max_row]);

        for (int j = i + 1; j < n; j++) {
            double factor = A[j][i] / A[i][i];
            for (int k = i; k < n; k++) {
```

```
                A[j][k] -= factor * A[i][k];
            }
            b[j] -= factor * b[i];
        }
    }

    for (int i = n - 1; i >= 0; i--) {
        x[i] = b[i];
        for (int j = i + 1; j < n; j++) {
            x[i] -= A[i][j] * x[j];
        }
        x[i] /= A[i][i];
    }
    // Wyświetlenie danych oraz wpisanie wyników do plików
}
```

Metoda solve_system() rozwiązuje układ równań liniowych postaci: A * T_global = b. Pętla w funkcji wykonuje eliminację Gaussa w celu przekształcenia macierzy do postaci schodkowej. Działa to na zasadzie zamiany wierszy i aktualizacji współczynników w macierzy A oraz wektorze b. Dla każdego wiersza, maksymalny element w kolumnie jest wybierany (tzw. pivoting), co pomaga zminimalizować błąd numeryczny. Po przekształceniu macierzy do formy schodkowej, rozwiązanie układu równań jest uzyskiwane za pomocą podstawienia wstecznego. Dla każdej zmiennej (w tym przypadku temperatury w węzłach), oblicza się jej wartość na podstawie wcześniejszych wartości.

3. Przedstawienie wyników

Wyniki temperatur w czasie dla siatki 4x4:

```
Simulation results:

Time: 50.0000000000000 s
Minimum Temperature: 110.0379723555507
Maximum Temperature: 365.8154726251593

Time: 100.0000000000000 s
Minimum Temperature: 168.8370097662479
Maximum Temperature: 502.5917142786477

Time: 150.0000000000000 s
Minimum Temperature: 242.8008462722100
Maximum Temperature: 587.3726667096676

Time: 200.0000000000000 s
Minimum Temperature: 318.6145887045103
Maximum Temperature: 649.3874821805224

Time: 250.0000000000000 s
Minimum Temperature: 391.2557917894920
```

```
Maximum Temperature: 700.0684182944657

Time: 300.0000000000000 s
Minimum Temperature: 459.0369089191098
Maximum Temperature: 744.0633414735055

Time: 350.0000000000000 s
Minimum Temperature: 521.5862853956570
Maximum Temperature: 783.3828462176099

Time: 400.0000000000000 s
Minimum Temperature: 579.0344613923548
Maximum Temperature: 818.9921835720459

Time: 450.0000000000000 s
Minimum Temperature: 631.6892582329691
Maximum Temperature: 851.4310377963708

Time: 500.0000000000000 s
Minimum Temperature: 679.9076191303863
Maximum Temperature: 881.0576293885946
```

Wyniki temperatur w czasie dla siatki 31x31:

```
Simulation results:

Time: 1.0000000000000 s
Minimum Temperature: 100.0000000002718
Maximum Temperature: 149.5569518081163

Time: 2.0000000000000 s
Minimum Temperature: 100.0000000052910
Maximum Temperature: 177.4449279500686

Time: 3.0000000000000 s
Minimum Temperature: 100.0000000514605
Maximum Temperature: 197.2669629217000

Time: 4.0000000000000 s
Minimum Temperature: 100.0000003344194
Maximum Temperature: 213.1527872915314

Time: 5.0000000000000 s
Minimum Temperature: 100.0000016382405
Maximum Temperature: 226.6825834190759

Time: 6.0000000000000 s
Minimum Temperature: 100.0000064712036
```

```
Maximum Temperature: 238.6070648058806

Time: 7.0000000000000 s
Minimum Temperature: 100.0000215293651
Maximum Temperature: 249.3466919424991

Time: 8.0000000000000 s
Minimum Temperature: 100.0000622127409
Maximum Temperature: 259.1650791551302

Time: 9.0000000000000 s
Minimum Temperature: 100.0001597833876
Maximum Temperature: 268.2406890050142

Time: 10.0000000000000 s
Minimum Temperature: 100.0003713449195
Maximum Temperature: 276.7010978633191

Time: 11.0000000000000 s
Minimum Temperature: 100.0007922355066
Maximum Temperature: 284.6412831886670

Time: 12.0000000000000 s
Minimum Temperature: 100.0015698461014
Maximum Temperature: 292.1342190508954

Time: 13.0000000000000 s
Minimum Temperature: 100.0029174838382
Maximum Temperature: 299.2374099453061

Time: 14.0000000000000 s
Minimum Temperature: 100.0051267975284
Maximum Temperature: 305.9971215275228

Time: 15.0000000000000 s
Minimum Temperature: 100.0085774636276
Maximum Temperature: 312.4512302135302

Time: 16.0000000000000 s
Minimum Temperature: 100.0137432142323
Maximum Temperature: 318.6312061364377

Time: 17.0000000000000 s
Minimum Temperature: 100.0211937597087
Maximum Temperature: 324.5635314899432

Time: 18.0000000000000 s
Minimum Temperature: 100.0315926140679
Maximum Temperature: 330.2707391733735
```

```
Time: 19.0000000000000 s
Minimum Temperature: 100.0456912009781
Maximum Temperature: 335.7721890479794

Time: 20.0000000000000 s
Minimum Temperature: 100.0643198699039
Maximum Temperature: 341.0846585343211
```

Wyniki temperatur w czasie dla siatki 4x4mix:

```
Simulation results:

Time: 50.0000000000000 s
Minimum Temperature: 95.1589832710790
Maximum Temperature: 374.6682824909749

Time: 100.0000000000000 s
Minimum Temperature: 147.6557663274260
Maximum Temperature: 505.9542722984855

Time: 150.0000000000000 s
Minimum Temperature: 220.1779701377777
Maximum Temperature: 586.9894318549053

Time: 200.0000000000000 s
Minimum Temperature: 296.7507266971076
Maximum Temperature: 647.2801205357165

Time: 250.0000000000000 s
Minimum Temperature: 370.9825026970679
Maximum Temperature: 697.3298722193082

Time: 300.0000000000000 s
Minimum Temperature: 440.5738838409430
Maximum Temperature: 741.2156525786797

Time: 350.0000000000000 s
Minimum Temperature: 504.9042571063853
Maximum Temperature: 781.2408551735243

Time: 400.0000000000000 s
Minimum Temperature: 564.0138159696068
Maximum Temperature: 817.4205119847389

Time: 450.0000000000000 s
Minimum Temperature: 618.1853991083369
Maximum Temperature: 850.2641124942197
```

```
Time: 500.0000000000000 s
Minimum Temperature: 667.7763493799891
Maximum Temperature: 880.1923052430849
```

Wyniki dla testu niestacjonarnego:

Pierwsza część testu:

Macierz C:

results > global_C_matrix.txt

```
Global C matrix:

674.07404 337.03702 0.00000 0.00000 337.03702 168.51851 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
337.03702 1348.14817 337.03706 0.00000 168.51851 674.07408 168.51853 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 337.03706 1348.14825 337.03706 0.00000 168.51853 674.07413 168.51853 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 337.03706 674.07413 0.00000 0.00000 168.51853 337.03706 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
337.03702 168.51851 0.00000 0.00000 1348.14806 674.07403 0.00000 0.00000 337.03701 168.51851 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
168.51851 674.07408 168.51853 0.00000 674.07403 2696.29629 674.07412 0.00000 168.51851 674.07406 168.51853 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 168.51853 674.07413 168.51853 0.00000 674.07412 2696.29646 674.07412 0.00000 168.51853 674.07411 168.51853 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 168.51853 337.03706 0.00000 0.00000 674.07412 1348.14823 0.00000 0.00000 168.51853 337.03705 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 337.03701 168.51851 0.00000 0.00000 1348.14801 674.07400 0.00000 0.00000 337.03699 168.51850 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 168.51851 674.07406 168.51853 0.00000 674.07400 2696.29618 674.07409 0.00000 168.51850 674.07403 168.51852 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 168.51853 674.07411 168.51853 0.00000 674.07409 2696.29635 674.07409 0.00000 168.51852 674.07407 168.51852
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 168.51853 337.03705 0.00000 0.00000 674.07409 1348.14818 0.00000 0.00000 168.51852 337.03703
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 337.03699 168.51850 0.00000 0.00000 674.07398 337.03699 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 168.51850 674.07403 168.51852 0.00000 337.03699 1348.14805 337.03703 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 168.51852 674.07407 168.51852 0.00000 337.03703 1348.14814 337.03703
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 168.51852 337.03703 0.00000 0.00000 337.03703 674.07407
```

Macierz H:

results > global_Hbc_matrix.txt

```
Global Hbc matrix:

16.66667 -4.16667 0.00000 0.00000 -4.16667 -8.33333 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
-4.16667 33.33333 -4.16667 0.00000 -8.33333 -8.33333 -8.33333 -8.33333 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 -4.16667 33.33333 -4.16667 0.00000 -8.33333 -8.33333 -8.33333 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 -4.16667 16.66667 0.00000 0.00000 -8.33333 -4.16667 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
-4.16667 -8.33333 0.00000 0.00000 33.33333 -8.33334 0.00000 0.00000 -4.16667 -8.33333 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
-8.33333 -8.33333 -8.33333 0.00000 -8.33334 66.66667 -8.33333 0.00000 -8.33333 -8.33333 -8.33333 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 -8.33333 -8.33333 -8.33333 0.00000 -8.33333 66.66667 -8.33333 0.00000 -8.33333 -8.33333 -8.33333 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 -8.33333 -4.16667 0.00000 0.00000 -8.33333 33.33333 0.00000 0.00000 -8.33333 -4.16667 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 -4.16667 -8.33333 0.00000 0.00000 33.33333 -8.33333 0.00000 0.00000 -4.16667 -8.33333 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 -8.33333 -8.33333 -8.33333 0.00000 -8.33333 66.66667 -8.33333 0.00000 -8.33333 -8.33333 -8.33333 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 -8.33333 -8.33333 -8.33333 0.00000 -8.33333 66.66667 -8.33333 0.00000 -8.33333 -8.33334 -8.33333
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -8.33333 -4.16667 0.00000 0.00000 -8.33333 33.33333 0.00000 0.00000 -8.33333 -4.16667
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -4.16667 -8.33333 0.00000 0.00000 16.66667 -4.16667 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -8.33333 -8.33333 -8.33333 0.00000 -4.16667 33.33333 -4.16667 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -8.33333 -8.33334 -8.33333 0.00000 -4.16667 33.33333 -4.16667
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -8.33333 -4.16667 0.00000 0.00000 -4.16667 16.66667

```

Macierz H = H + C / dT i P = P + {C / dT} * {T0}:

Iteracja 0:

```
-----------------------------------
Matrix [H] + [C]/dT at time: 50 s
36.81481 4.24074 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
4.24074 66.96296 4.24074 0.00000 -4.96296 5.14815 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 4.24074 66.96297 4.24074 0.00000 -4.96296 5.14815 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 4.24074 36.81482 0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
4.24074 -4.96296 0.00000 0.00000 66.96296 5.14815 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
-4.96296 5.14815 -4.96296 0.00000 5.14815 120.59259 5.14815 0.00000 -4.96296 5.14815 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 -4.96296 5.14815 -4.96296 0.00000 5.14815 120.59260 5.14815 0.00000 -4.96296 5.14815 -4.96296 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 5.14815 66.96296 0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000 66.96296 5.14815 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 -4.96296 5.14815 -4.96296 0.00000 5.14815 120.59259 5.14815 0.00000 -4.96296 5.14815 -4.96296 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 5.14815 -4.96296 0.00000 5.14815 120.59259 5.14815 0.00000 -4.96296 5.14815 -4.96296
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 5.14815 66.96296 0.00000 0.00000 -4.96296 4.24074
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000 36.81481 4.24074 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 5.14815 -4.96296 0.00000 4.24074 66.96296 4.24074 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 5.14815 -4.96296 0.00000 4.24074 66.96296 4.24074
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 4.24074 36.81481
Vector P ([{P} + {[C]/dT}*{T0}]) at time: 50.00000 s
15033.33291 18066.66661 18066.66775 15033.33404 18066.66642 12133.33333 12133.33409 18066.66718 18066.66564 12133.33280 12133.33356 18066.66640 15033.33212 18066.66609 18066.66723 15033.33326
-----------------------------------
```

Iteracja 1:

```
-----------------------------------
Matrix [H] + [C]/dT at time: 100.00000 s
36.81481 4.24074 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
4.24074 66.96296 4.24074 0.00000 -4.96296 5.14815 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 4.24074 66.96297 4.24074 0.00000 -4.96296 5.14815 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 4.24074 36.81482 0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
4.24074 -4.96296 0.00000 0.00000 66.96296 5.14815 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
-4.96296 5.14815 -4.96296 0.00000 5.14815 120.59259 5.14815 0.00000 -4.96296 5.14815 -4.96296 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 -4.96296 5.14815 -4.96296 0.00000 5.14815 120.59260 5.14815 0.00000 -4.96296 5.14815 -4.96296 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 5.14815 66.96296 0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000 66.96296 5.14815 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 -4.96296 5.14815 -4.96296 0.00000 5.14815 120.59259 5.14815 0.00000 -4.96296 5.14815 -4.96296 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 5.14815 -4.96296 0.00000 5.14815 120.59259 5.14815 0.00000 -4.96296 5.14815 -4.96296
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 5.14815 66.96296 0.00000 0.00000 -4.96296 4.24074
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 4.24074 -4.96296 0.00000 0.00000 36.81481 4.24074 0.00000 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 5.14815 -4.96296 0.00000 4.24074 66.96296 4.24074 0.00000
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 5.14815 -4.96296 0.00000 4.24074 66.96296 4.24074
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 -4.96296 4.24074 0.00000 0.00000 4.24074 36.81481
Vector P ([{P} + {[C]/dT}*{T0}]) at time: 100.00000 s
20659.69828 25552.22418 25552.22566 20659.69998 25552.22359 18897.39137 18897.39228 25552.22500 25552.22258 18897.39075 18897.39165 25552.22398 20659.69710 25552.22322 25552.22469 20659.69881
-----------------------------------
```

Wyniki temperatur zostały przedstawione już w teście siatki 4x4.

Druga część testu:

Wyniki temperatur zostały przedstawione już w teście siatki 31x31.

Wyniki dla testu stacjonarnego:

```
-----------------------------------
Global t vector:

1200 1200 1200 1200 1200 1200 1200 1200 1200 1200 1200 1200 1200 1200 1200 1200
```