

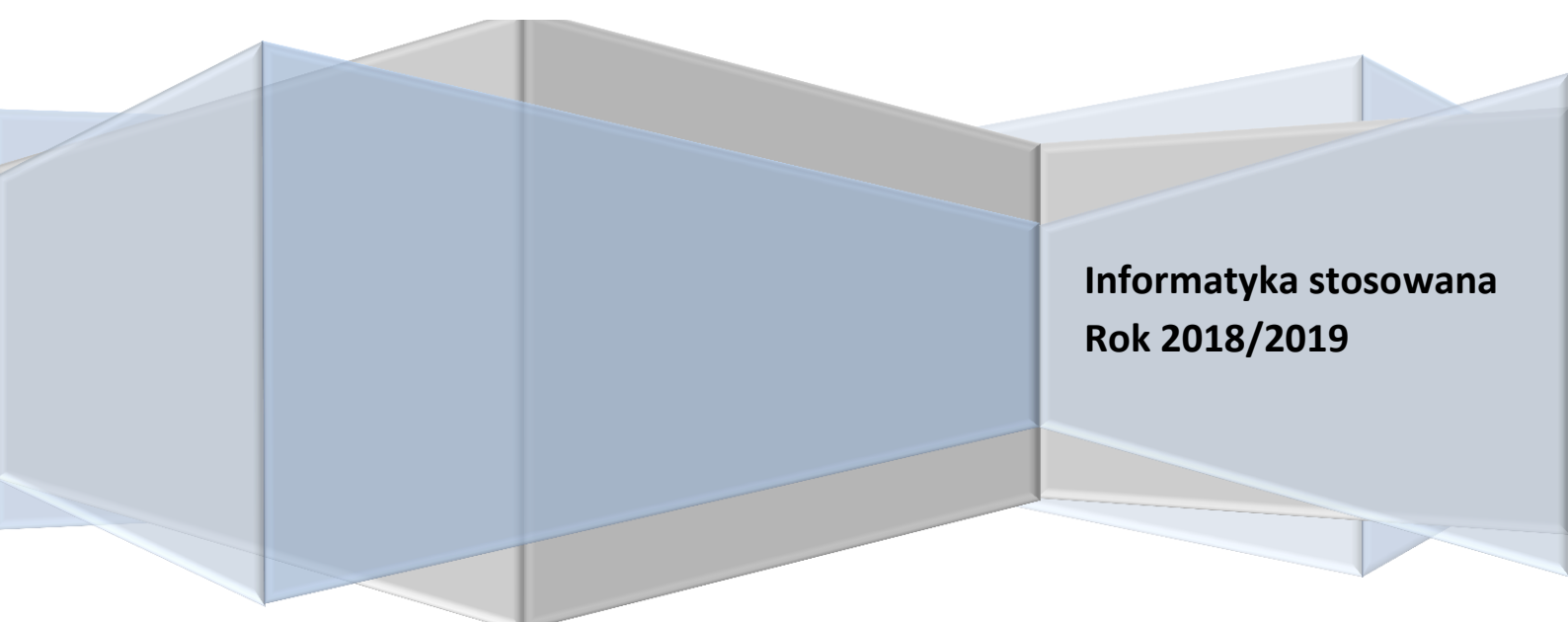
Metoda Elementów Skończonych

Sprawozdanie

Autor: Przemysław Szymoniak

3 rok IS IMiP AGH

Nr Indeksu: 293250



**Informatyka stosowana
Rok 2018/2019**

Spis treści

Wstęp	2
Struktura aplikacji.....	3
Struktura folderów i plików - krótki opis.....	3
Struktura pliku z danymi.....	4
Diagram klas pomocniczych - nie wyliczających macierzy i wektorów	4
Diagram klas głównych - obliczających macierze i wektory	5
Opis Programu.....	6
Generowanie siatki MES.....	6
• Klasa Grid.....	6
• Klasa Node	6
• Klasa Element	7
Obliczanie macierzy lokalnych.....	8
• Macierze Jakobiego - klasa jakobian2D.....	8
• Macierz H - klasa matrixH.....	9
• Macierz warunku brzegowego H - klasa matrixH_BC.....	9
• Macierz C - klasa matrixC	10
Obliczenie wektorów.....	11
• Wektor P - klasa VectP	11
Agregacja Macierzy i Wektorów.....	12
• Globalna macierz H - klasa globalMatrixH.....	12
• Globalna macierz C - klasa globalMatrixC	12
• Globalny wektor P - klasa Vecp	13
Wyliczanie wektora temperatur.....	13
Testy Aplikacji.....	15

Wstęp

Cały dokument został poświęcony opisaniu programu napisanego przeze mnie wykonującego obliczenia z wykorzystaniem Metody elementów skończonych. W pierwotnej wersji aplikacji całość miała wykorzystywać trójwymiarowe tablice stworzone dynamicznie. Jednak w trakcie prac nad zaimplementowaniem algorytmów okazało się, że nie jest to zbyt rozsądny pomysł ponieważ w tej wersji bardzo łatwo o pomyłkę. Próba wyeliminowania takiego błędu potrafi zająć dłuższą chwilę. Dlatego też całość przy użyciu wcześniej napisanego kodu została przerobiona aby móc wykorzystywać pobraną z Internetu klasę „matrix”. Pomimo iż aplikacja w każdej klasie będącej macierzą (np. dla macierzy H) powinna dziedziczyć „matrix”, mój kod tego nie wykorzystuje z kilku powodów:

- ✓ początkowa przeróbka z tablic dynamicznych została na szybko podmieniona na macierz,
- ✓ każda klasa ma być jedynie pewnego rodzaju pojemnikiem przechowującym jedynie operacje związane z obliczaniem poszczególnych składowych programu,
- ✓ cały kod miał w najprostszy sposób pomóc zrozumieć mechanizmy obliczania poszczególnych wektorów i macierzy
- ✓ kod pozwalający na szybkie załapanie jego działania nawet po dłuższej przerwie w jego pisaniu.

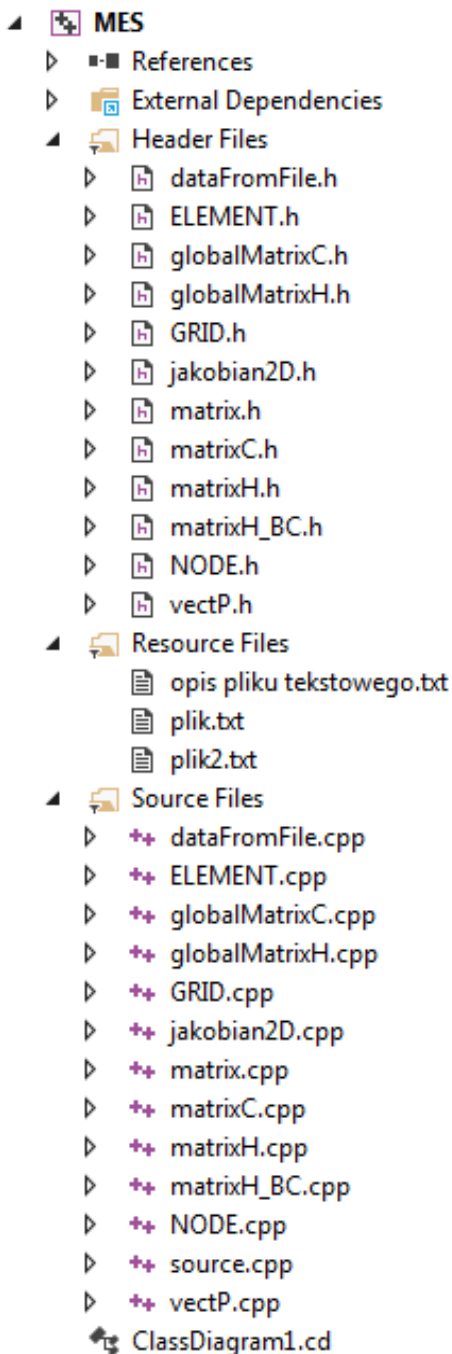
Dodatkowo dodałem do kodu dużą liczbę komentarzy, która pozwala szybciej odnaleźć się w działaniu programu. Każda poszczególna część wymagająca obliczenia znajduje się w innej klasie z góry pozwalając na chociaż częściowe przyspieszenie wyszukiwania szczególnie interesujących nas informacji.

Cały kod został napisany przy użyciu języka C++ oraz Microsoft Visual Studio. Początkowo była to wersja VS 2010 Express która jest wyjątkowo lekką i szybką aplikacją pozwalającą uruchomić się nawet na nie wymagającym sprzęcie kończą na VS 2015 Enterprise posiadającym możliwość generowania diagramu klas.

Do zrealizowania programu wykorzystano wiedzę uzyskaną na zajęciach, dane z arkusza kalkulacyjnego Excel: „**Jakobian2d_excel**” oraz informacje zawarte w plikach „*.pdf” na stronie prowadzącego zajęcia.

Struktura aplikacji

Struktura folderów i plików - krótki opis



Klasy:

- ✓ Matrix to klasa dostarczająca tworzenie macierzy oraz podstawowe operacje na niej np. wprowadzanie danych, mnożenie itp.
- ✓ DataFromFile jest odpowiedzialna za pobranie i przechowywanie informacji zawartych w pliku tekstowym
- ✓ Element jest klasą odzwierciedlającą pojedynczy element siatki MES przechowując o nim istotne informacje
- ✓ Node jest klasą odpowiedzialną za jeden pojedynczy węzeł siatki MES
- ✓ Grid to klasa przechowująca zbiór elementów oraz węzłów, jednocześnie na podstawie danych z pliku generuje siatkę MES, oznaczając konkretne węzły jako brzegowe
- ✓ GlobalMatrixC to klasa będąca odzwierciedleniem globalnej macierzy C, posiada metody umożliwiającą agregację oraz takie wymagane do obliczenia wektora temperatur
- ✓ GlobalMatrixH to klasa będąca odzwierciedleniem globalnej macierzy H, posiada metody umożliwiającą agregację oraz takie wymagane do obliczenia wektora temperatur
- ✓ Jakobian2D to klasa która oblicza macierz jacobiego dla jednego elementu, na jej podstawie określając poszczególne jacobiany
- ✓ MatrixC jest klasą zawierającą algorytm potrzebny do wyliczenia lokalnej macierzy C dla jednego elementu wraz z metodami wymaganymi do agregacji
- ✓ MatrixH jest klasą zawierającą algorytm potrzebny do wyliczenia lokalnej macierzy H dla jednego elementu wraz z metodami wymaganymi do agregacji
- ✓ Matrix_BC to klasa wyliczająca lokalne warunki brzegowe dla pojedynczego elementu z wykorzystaniem całkowania 1D wraz z metodami potrebnymi do agregacji.
- ✓ VectP to klasa wyliczająca warunki brzegowe przy użyciu całkowania 1D oraz metody agregacji

Struktura pliku z danymi

Każda oddzielna linia traktowana jest jako pojedyncza dana wejściowa według schematu:

1. Czas Symulacji
2. Co jaki krok zmieniamy czas
3. Temperatura otoczenia
4. Temperatura początkowa
5. Współczynnik przewodności cieplnej
6. Współczynnik gęstości materiału
7. Ciepło właściwe materiału
8. Wysokość siatki MES (H):
9. Szerokość siatki MES (L)
10. Liczba pionowych linii siatki (nh)
11. Liczba poziomych linii siatki (nl)
12. Współczynnik przenikania ciepła (alfa)

Diagram klas pomocniczych - nie wyliczających macierzy i wektorów

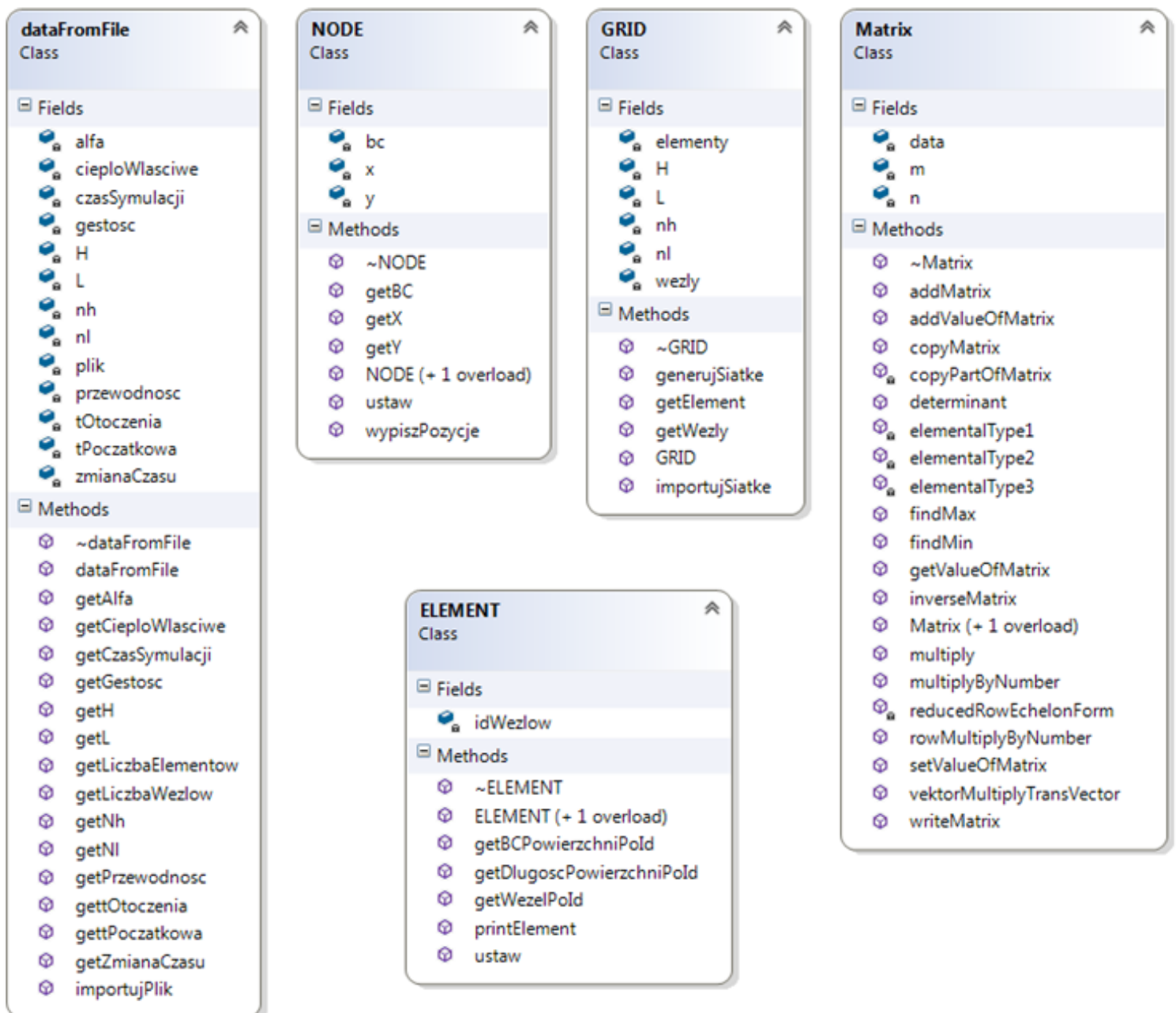
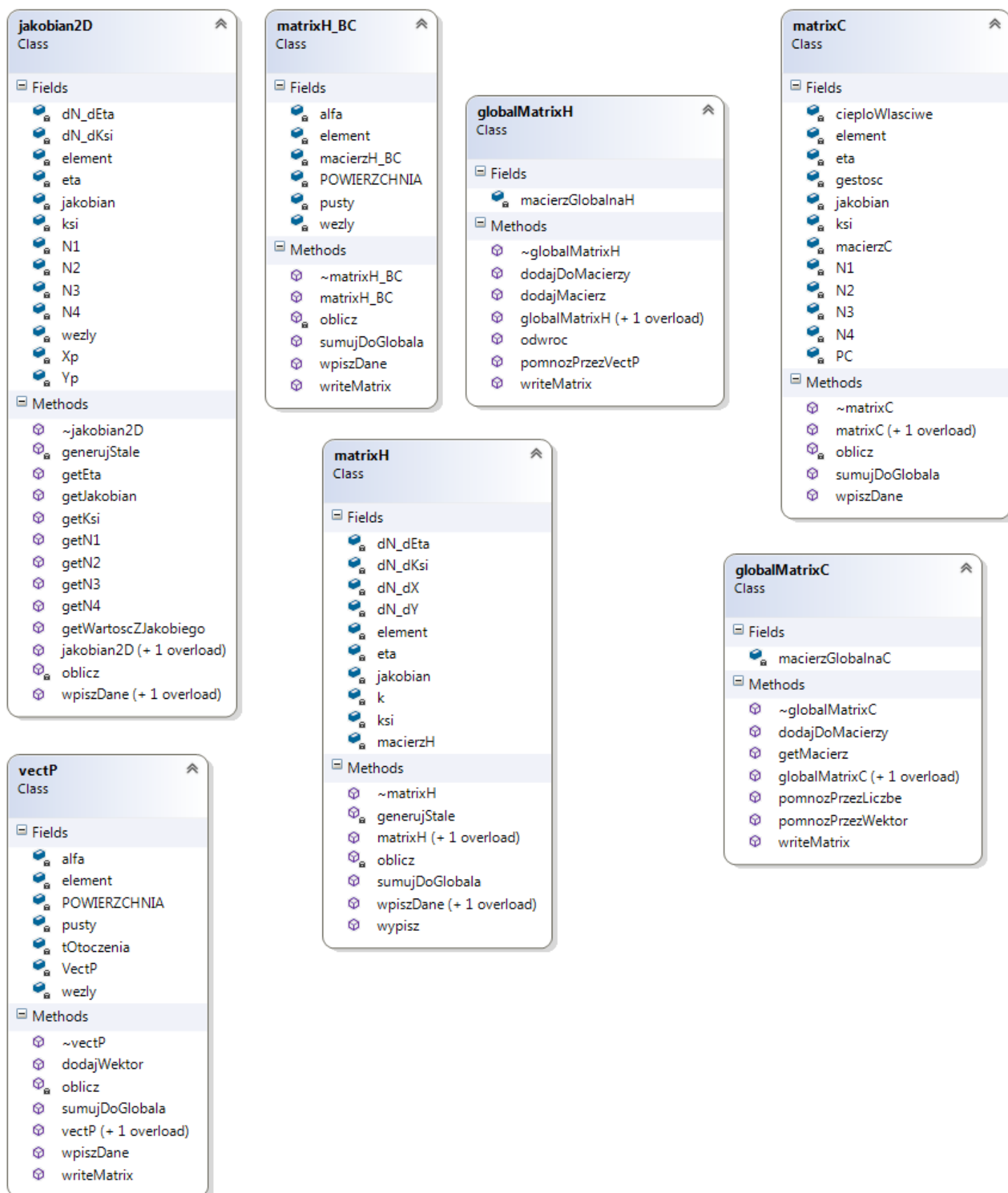


Diagram klas głównych - obliczających macierze i wektory



Opis Programu

Generowanie siatki MES

• Klasa Grid

Aby móc wygenerować siatkę MES po utworzeniu zaimportowaniu zewnętrznego pliku tekstowego z danymi należy utworzyć obiekt klasy Grid a następnie wykorzystać metodę `generujSiatke(dataFromFile* DaneZPliku)`. Metoda ta w pierwszej kolejności pobierze interesujące ją informacje jak *wysokość i szerokość siatki* oraz *liczbę linii pionowych oraz poziomych siatki*. Jeśli wszystkie dane są poprawne rozpoczynamy proces tworzenia siatki.

Dynamicznie tworzymy tablice elementów o wymiarze będącej liczbą elementów oraz tablice węzłów o wymiarze będącym liczbą węzłów. W pierwszej kolejności ustawiamy dane poszczególnych węzłów wykorzystując metodę `NODE::ustaw(double X, double Y, bool BC)`. W pętli sprawdzamy czy węzeł występuje na brzegu siatki ustawiając odpowiednio flagę `bool BC` oraz określając pozycję **(X,Y)** konkretnych węzłów na podstawie licznika pętli dodając odpowiednią ilość razy ΔX i ΔY .

W następnym ruchu przyporządkowujemy określonemu elementowi poszczególne węzły siatki. Wykorzystując metodę `ELEMENT::ustaw(int pierwszy, int nH)`.

Klasa Grid przechowuje takie zmienne jak:

- ✓ Dynamicznie utworzoną tablicę węzłów
- ✓ Dynamicznie utworzoną tablicę elementów
- ✓ Wysokość i szerokość siatki MES
- ✓ Liczba pionowych oraz poziomych linii siatki MES

Pozostałe metody klasy Grid:

- ✓ `ELEMENT * getElement(int numer)` – zwraca element o id <0; Liczba elementów>
- ✓ `void importujSiatke(ELEMENT * el, NODE * no)` – pozwala wykorzystywać ręcznie wygenerowane tablice elementów i węzłów
- ✓ `NODE * getWezly()` – zwraca tablicę wszystkich węzłów.

• Klasa Node

Jest to klasa nie wykonywująca żadnych istotnych obliczeń, przechowuje jedynie informacje o pozycji oraz istnieniu warunku brzegowego w konkretnym węźle siatki MES.

Klasa Node przechowuje takie zmienne jak:

- ✓ Współrzędna X
- ✓ Współrzędna Y
- ✓ Flaga obecności warunku brzegowego(BC)

Metody klasy Node:

- ✓ `void ustaw(double, double, bool)` – Pozwala wpisać dane
- ✓ `void wypiszPozycje(int id)` – wypisuje informacje na podstawie identyfikatora
- ✓ `double getY(), getX(), bool getBC()` – zwracają informacje przechowywane w klasie

- **Klasa Element**

Podobnie do klasy Node nie wykonuje w niej żadnych obliczeń, służy jedynie do przechowywania informacji.

Klasa Element przechowuje takie zmienne jak:

- ✓ 4 elementowa tablica w której przechowujemy kolejne węzły elementu.

Metody klasy Element:

- ✓ `void ustaw(int pierwszy, int nH)` – służy do ustawienia konkretnych węzłów do elementu
- ✓ `int getWezelPoId(int ID)` – zwraca identyfikator konkretnego węzła przy użyciu ID(0-3) lokalnego wykorzystywanego w pojedynczym elemencie
- ✓ `bool getBCPowierzchniPoId(NODE *wezly, int ID)` – metoda zwraca prawdę w sytuacji obecności warunku brzegowego na powierzchni o ID(0-3) wykorzystuje przy tym globalną tablicę `wezly` oraz identyfikatory węzłów wpisane do elementu.
- ✓ `double getDlugoscPowierzchniPoId(NODE *wezly, int ID)` – metoda zwraca długość powierzchni o ID(0-3) obliczana przy pomocy wzoru matematyczne na długość odcinka w układzie kartezjańskim.
- ✓ `void printElement()` – wypisuje wszystkie zapisane informacje o elemencie.

Obliczanie macierzy lokalnych

- **Macierze Jakobiego - klasa jakobian2D**

Jest to klasa odpowiedzialna za wyliczenie jacobianu przekształcenia dla pojedynczego elementu. Tak naprawdę obliczamy 4 macierze jacobiego o wymiarze 2x2. Wyznacznik każdej z macierzy jest jacobianem dla odpowiedniego punktów całkowania.

Pierwszym w kolejności krokiem do wykonywania obliczeń jest wyznaczenie stałych w metodzie `void jakobian2D::generujStale()` oblicza ona wartości κ/η , funkcji kształtu w punktach całkowania na ich podstawie obliczając X_p i Y_p dla każdego punktu całkowania. Następnie obliczamy pochodne $dN/d\kappa$ oraz $dN/d\eta$ (wykorzystując wzory na pochodne $dN/d\kappa$, $dN/d\eta$ oraz wartości η/κ w punktach całkowania) dla wszystkich czterech funkcji kształtu uzyskując 2 macierze 4x4. Następnie wyliczamy macierze jacobiego mnożąc poszczególne pola macierzy $dN/d\kappa$ oraz $dN/d\eta$ przez odpowiednie współrzędne węzłów siatki uzyskując 4 macierze 2x2.

Klasa jakobian2D przechowuje takie zmienne jak:

- ✓ Wskaźnik do elementu dla którego wyliczane są macierze jacobiego
- ✓ Wskaźnik do tablicy zawierającej listę węzłów
- ✓ Lokalne macierze przechowujące wartości funkcji kształtu oraz κ/η dla konkretnych punktów całkowania
- ✓ Lokalne macierze $dN/d\eta$ oraz $dN/d\kappa$
- ✓ Tablice 4 macierzy zawierającej jacobiany dla poszczególnych punktów całkowania

Metody klasy Element:

- ✓ `void wpiszDane(ELEMENT*, NODE *)` – umożliwia wpisanie potrzebnych danych przy użyciu wskaźnika do określonego elementu oraz wskaźnika do tablicy wszystkich węzłów
- ✓ `void wpiszDane(GRID* siatka, int numerElementu)` – służy do wprowadzania wszystkich danych przy użyciu danych z wygenerowanej siatki MES oraz konkretnego numeru elementu
- ✓ `Matrix *getKsi()` oraz `Matrix *getEta()` – zwraca wskaźniki do tablic zawierających wartości κ/η w określonych punktach całkowania
- ✓ `Matrix *getN1(), getN2(), getN3(), *getN4()` – zwraca wskaźniki do tablic zawierających wartości funkcji kształtu w punktach całkowania
- ✓ `double getWartoscZJakobiego(int punktCalkowania, int n, int m)` – dla konkretnego punktu całkowania zwraca liczbę o współrzędnych $[n, m]$ z macierzy jacobiego pomnożony przez jacobian
- ✓ `double getJakobian(int punktCalkowania)` – zwraca liczbę będącą wyznacznikiem macierzy jacobiego dla określonego punktu całkowania

- **Macierz H - klasa matrixH**

Jest to klasa wyliczająca lokalną macierz pojemności cieplnej o wymiarze 4x4 dla pojedynczego elementu siatki MES.

Pierwszym w kolejności krokiem do wykonywania obliczeń jest wyznaczenie stałych w metodzie `void matrixH::generujStale()` oblicza ona 2 macierze 4x4 zawierające wartości dla pochodnych $dN/dksi$ oraz $dN/deta$. Następnie wyliczamy 2 macierze 4x4 dN/dX oraz dN/dY wykorzystując wcześniej wygenerowane stałe macierze pomnożone przez odpowiednie wartości zwrócone z jakobianu metodą `double getWartoscZJakobiego(...)` zsumowane ze sobą tworzą odpowiednio macierz dN/dX oraz dN/dY . Kolejnym ruchem jest pomnożenie wiersza dN/dX i dN/dY przez ten sam wiersz transponowany. Uzyskane 8 macierzy 4x4 mnożymy przez współczynnik przewodności(k) oraz jacobian. Sumując wszystkie 9 macierzy uzyskujemy jedną macierz H o wymiarze 4x4.

Klasa matrixH przechowuje takie zmienne jak:

- ✓ Wskaźnik do elementu dla którego wyliczane są lokalne macierze H
- ✓ Wskaźnik do jacobian2D obliczonego na podstawie tego samego elementu
- ✓ Wskaźniki do macierzy przechowujących wartości funkcji kształtu oraz ksi/eta dla konkretnych punktów całkowania zwróconych przez jacobian2D
- ✓ Lokalne macierze $dN/deta$ oraz $dN/dksi$
- ✓ Lokalne macierze dN/dX oraz dN/dY
- ✓ Współczynnik przewodności cieplnej(k)
- ✓ Macierz H

Metody klasy matrixH:

- ✓ `void wpiszDane(jacobian2D *, double przewodność)` – umożliwia wpisanie potrzebnych danych przy użyciu wskaźnika do określonego jacobian2D oraz współczynnika przewodności cieplnej
- ✓ `void wpiszDane(GRID *siatka, jacobian2D * tablicaJacobianow, dataFromFile *daneZPliku, int numerElementu)` – służy do wprowadzania wszystkich danych przy użyciu danych z wygenerowanej siatki MES, wskaźnika do tablicy jacobianów, wskaźnika do klasy zawierającej dane z pliku oraz konkretnego numeru elementu
- ✓ `void sumujDoGlobala(globalMatrixH* globalnaMach)` – metoda odbierająca wskaźnik do globalnej macierzy H wykonująca na niej operacje niezbędne do agregacji macierzy
- ✓ `void wypisz()` – Wypisuje lokalną macierz H

- **Macierz warunku brzegowego H - klasa matrixH_BC**

Jest to klasa pozwalająca wprowadzić warunek brzegowy do macierzy H. Warunki są obliczane z wykorzystaniem całkowania 1D dla poszczególnych powierzchni w konkretnym elemencie. Uzyskane w ten sposób macierze 2x2 są następnie agregowane do lokalnej macierzy H_BC o wymiarze 4x4

Pierwszym krokiem jest sprawdzenie obecności warunku brzegowego w celu uniknięcia niepotrzebnych obliczeń. Następnie wyliczamy 2 wektory na podstawie funkcji kształtu oraz wartości w punkcie ksi. Później wyliczamy 2 macierze 2x2 mnożąc wektor funkcji kształtu przez transponowanego siebie. Sumujemy uzyskane macierze i mnożymy razy długość powierzchni uzyskaną(`ELEMENT::getDlugoscPowierzchniPoId(...)`) podzielony przez 2(Jacobian dla 1D)

Klasa matrixH_BC przechowuje takie zmienne jak:

- ✓ Wskaźnik do elementu dla którego wyliczane są lokalne macierze warunku brzegowego H
- ✓ Wskaźnik do tablicy zawierającej listę węzłów
- ✓ Współczynnik przenikania ciepła (alfa)
- ✓ Flaga pusty odwzorowujący obecność jakiegokolwiek warunku brzegowego
- ✓ Tablica[4] flag odnoszących się do obecności warunku na konkretnej powierzchni

- **Macierz C - klasa matrixC**

Jest to klasa wykorzystywana do obliczenia lokalnej macierzy własności materiałowych o wymiarach 4x4 dla pojedynczego elementu.

Wykorzystując wszystkie zmienne wygenerowane w klasie jakobian2D obliczamy poszczególne wartości mnożąc poszczególne funkcje kształtu razy siebie transponowane. Uzyskując 4 macierze 4x4 po jednej dla każdego punktu całkowania. Następnie wszystkie wartości w obliczonych macierzach mnożymy razy współczynnik gęstości oraz ciepło właściwe. Sumując 4 macierze uzyskujemy lokalną macierz C.

Klasa matrixC przechowuje takie zmienne jak:

- ✓ Wskaźnik do elementu dla którego wyliczane są lokalne macierze C
- ✓ Wskaźnik do jakobian2D obliczonego na podstawie tego samego elementu
- ✓ Wskaźniki do macierzy przechowujących wartości funkcji kształtu oraz ksi/eta dla konkretnych punktów całkowania zwróconych przez jakobian2D
- ✓ Tablica[4] macierzy będąca obliczeniem współczynników macierzy C dla każdego z punktów całkowania
- ✓ Współczynnik gęstości i ciepło właściwe materiału

Metody klasy matrixC:

- ✓ `void wpiszDane(Grid* siatka, jakobian2D * tablicaJakobianow, dataFromFile* daneZPliku, int numerElementu)` - służy do wprowadzania wszystkich danych przy użyciu danych z wygenerowanej siatki MES, wskaźnika do tablicy jakobianów, wskaźnika do klasy zawierającej dane z pliku oraz konkretnego numeru elementu
- ✓ `void sumujDoGlobala(globalMatrixC* globalnaMacC)` – metoda odbierająca wskaźnik do globalnej macierzy C wykonująca na niej operacje niezbędne do agregacji macierzy

Obliczenie wektorów

- **Wektor P - klasa VectP**

Obliczenie wektora P jest niezbędnym wprowadzeniem warunku brzegowego wymaganym do obliczenia temperatur w następnym kroku iteracyjnym. Całość została wykonana przy użyciu całkowania 1D.

Pierwszym krokiem jest sprawdzenie obecności warunku brzegowego w celu uniknięcia niepotrzebnych obliczeń. Następnie wyliczamy 2 wektory na podstawie funkcji kształtu oraz wartości w punkcie ksi. Następnie sumujemy do określonego miejsca w wektorze zależnie od punktu całkowania funkcje kształtu pomnożoną razy współczynnik przenikania ciepła, temperaturę otoczenia oraz jacobian 1D (długość powierzchni[(ELEMENT::getDlugoscPowierzchniPoId(..))]/2) operacje wykonywamy dla obydwu funkcji kształtu dla wszystkich punktów w których występuje warunek brzegowy.

Klasa vectP przechowuje takie zmienne jak:

- ✓ Wskaźnik do elementu dla którego wyliczane są wektory P
- ✓ Wskaźnik do tablicy zawierającej listę węzłów
- ✓ Współczynnik przenikania ciepła (alfa)
- ✓ Wartość temperatury otoczenia
- ✓ Flagę pusty eliminującą zbędne obliczenia przy braku warunku brzegowego
- ✓ Tablicę flag[4] odzwierciedlających występowanie warunku brzegowego na powierzchni

Metody klasy vectP:

- ✓ `void wpiszDane(Grid *siatka, jacobian2D * tablicaJakobianow, dataFromFile *daneZPliku, int numerElementu)` - służy do wprowadzania wszystkich danych przy użyciu danych z wygenerowanej siatki MES, wskaźnika do tablicy jacobianów, wskaźnika do klasy zawierającej dane z pliku oraz konkretnego numeru elementu
- ✓ `void sumujDoGlobala(vectP* globalnyVectP)` — metoda odbierająca wskaźnik do globalnego wektora P wykonuje na niej operacje niezbędne do agregacji wektora
- ✓ `void writeMatrix()` - wypisuje wektor P
- ✓ `void dodajWektor(Matrix * wektor)` - metoda dodaje do przechowywanego wektor przesłany przez wskaźnik o tym samym wymiarze

Agregacja Macierzy i Wektorów

• Globalna macierz H - klasa `globalMatrixH`

Większość operacji wykonywana jest wewnątrz macierzy H, jedynie samo wpisywanie do macierzy globalnej odbywa się wewnątrz klasy `globalMatrixH`.

Przesłany w argumencie do `matrixH` wskaźnik do globalnej macierzy H jest wykorzystywany przy agregacji. Wykorzystujemy 2 pętle pozwalające przelecieć przez wszystkie elementy lokalnej macierzy H. Następnie wykorzystujemy funkcję `element->getWezelPoId(..)` zwracającą globalny numer węzła przy użyciu lokalnego. Wprowadzamy do globalnej macierzy dane tłumacząc poszczególne współrzędne wewnątrz macierzy lokalnej na jej globalne odpowiedniki za pomocą: `globalnaMacH->dodajDoMacierzy(element->getWezelPoId(i),element->getWezelPoId(j), macierzH->getValueOfMatrix(i,j))`.

Na takiej samej zasadzie agregujemy warunki brzegowe z `matrixH_BC` sprawdzając wcześniej czy jest ustawiona flaga *pusty(agregacja przy braku)*, uzyskując kompletną macierz pojemności cieplnej.

Klasa `globalMatrixH` przechowuje takie zmienne jak:

- ✓ Macierz o wymiarach liczba węzłów x liczba węzłów

Metody klasy `globalMatrixH`:

- ✓ `void dodajDoMacierzy(int n, int m, double wartość)` – wpisuje poszczególne liczby w konkretne miejsce macierzy globalnej
- ✓ `void dodajMacierz(Matrix * macierz)` – jest funkcją pozwalającą na dokonywania sumowania macierzy przesłanej przez wskaźnik z macierzą przechowywaną w klasie
- ✓ `void writeMatrix()` – wypisuje macierz globalną H
- ✓ `void odwroc()` – odwraca macierz globalną
- ✓ `Matrix * pomnozPrzezVectP(vectP * wektorP)` – pozwala pomnożyć przechowywaną macierz przez wektor P zwracając uzyskaną macierz

• Globalna macierz C - klasa `globalMatrixC`

Większość operacji wykonywana jest wewnątrz macierzy C, jedynie samo wpisywanie do macierzy globalnej odbywa się wewnątrz klasy `globalMatrixC`.

Podobnie jak przy agregacji macierzy H korzystamy z 2 pętli pozwalających przelecieć przez wszystkie elementy lokalnej macierzy. Następnie wykorzystujemy funkcję `element->getWezelPoId(..)` zwracającą globalny numer węzła przy użyciu lokalnego. Wprowadzamy do globalnej macierzy dane tłumacząc poszczególne współrzędne wewnątrz macierzy lokalnej na jej globalne odpowiedniki za pomocą: `globalnaMacC->dodajDoMacierzy(element->getWezelPoId(i), element->getWezelPoId(j), macierzC->getValueOfMatrix(i, j))`.

Klasa `globalMatrixH` przechowuje takie zmienne jak:

- ✓ Macierz o wymiarach liczba węzłów x liczba węzłów

Metody klasy `globalMatrixH`:

- ✓ `void dodajDoMacierzy(int n, int m, double wartość)` – wpisuje poszczególne liczby w konkretne miejsce macierzy globalnej
- ✓ `void writeMatrix()` – wypisuje macierz globalną C
- ✓ `void pomnozPrzezLiczbe(double liczba)` – mnoży poszczególne wartości wewnątrz globalnej macierzy C przez liczbę przesłaną jako argument
- ✓ `Matrix * pomnozPrzezWektor(Matrix * wektor)` – mnoży przechowywaną macierz przez wektor przesłany jako wskaźnik
- ✓ `Matrix *getMacierz()` – zwraca wskaźnik do macierzy przechowywanej przez klasę.

- **Globalny wektor P - klasa Vectp**

Globalny wektor P jest obiektem klasy VectP tak samo jak jego lokalny odpowiednik. Różnicą jest jedynie wymiar, który dla lokalnej wersji wynosi 4 a dla globalnej równowartość ilości występujących elementów.

Podobnie jak w przypadku agregacji macierzy H_BC pierwszym krokiem jest sprawdzenie występowania flagi puste. W przypadku jej ustawienia operacja nie jest dokonywana ponieważ element nie posiada warunku brzegowego. W sytuacji jej braku zaczynamy proces agregacji wykorzystując pojedynczą pętlę oraz funkcję `element->getWezelPoId(..)` zwracającą globalny numer węzła przy użyciu lokalnego. Następnie wprowadzamy do globalnego wektora dane tłumacząc poszczególne współrzędne wewnątrz wektora lokalnego na jego globalny odpowiednik za pomocą:

```
globalnyVectP->VectP->setValueOfMatrix(element->getWezelPoId(kolumna), 0,
this->VectP->getValueOfMatrix(kolumna,0));
```

Klasa przechowuje takie same metody i pola jak wcześniej.

Wyliczanie wektora temperatur

Wektory temperatur są wyliczane w pliku source.c są w nim zawarte również wszystkie inicjalizacje oraz wywołania funkcji wymaganych do wykorzystania we wzorze:

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

Upraszczamy stosując:

$$[\hat{H}] = \left([H] + \frac{[C]}{\Delta\tau} \right), \quad \{\hat{P}\} = \left(\frac{[C]}{\Delta\tau} \right) \{t_0\} + \{P\}$$

Do postaci:

$$[\hat{H}] \{t_1\} + \{\hat{P}\} = 0$$

Przekształcając to równanie do postaci:

$$\{t_1\} = [\hat{H}]^{-1} * \{\hat{P}\}$$

Uzyskujemy sposób na wyliczenie wektora temperatur.

W kodzie [H] jest globalną macierzą H, [C] globalną macierzą C, {P} globalnym wektorem P a {t₀} wektorem temperatur węzłowych w pierwszej iteracji dla każdego węzła równa temperaturze początkowej.

Pierwszym krokiem jest obliczenie [C]/dT za pomocą funkcji:

```
globalnaMacC->pomnozPrzezLiczbe(1.0/(daneZPliku.getZmianaCzasu()));
```

Macierz globalna C została obliczona i będzie wykorzystywana w następnym kroku.

Obliczamy [H'] dodając globalną macierz H z wcześniej uzyskaną macierzą C (C/dT) za pomocą: `globalnaMacH->dodajMacierz(globalnaMacC->getMacierz());`

W tym momencie mamy gotową macierz [H'] = [H] + [C]/dT

Kolejną ważną sprawą będzie pomnożenie macierzy [C]*{T₀} (([C]/dT)*{T₀}) uzyskane dzięki: `Matrix* C_przez_dT_razyTemp = globalnaMacC->pomnozPrzezWektor(wektorTemp);`

Uzyskany w ten sposób wektora dodajemy do wektora P przy użyciu: `globalnyVectP->dodajWektor(C_przez_dT_razyTemp);`

W ten sposób przygotowaliśmy wszystko do obliczenia wektora temperatury. Obliczamy bo zgodnie ze wzorem wyjaśnionym na poprzedniej stronie.

Chcąc wykorzystać metodę macierzy odwrotnej oraz wzór:

$$\{t_1\} = [\hat{H}]^{-1} * \{\hat{P}\}$$

Najpierw odwracamy macierz [H'] z pomocą biblioteki do tego przeznaczonej:

```
globalnaMacH->odwroc();
```

A następnie wyliczamy nowy wektor temperatur za pomocą :

```
wektorTemp = globalnaMacH->pomnozPrzezVectP(globalnyVectP);
```

Chcąc uzyskać wektor temperatur w następnej iteracji podmieniamy nasz wektor temperatur na ten właśnie obliczony. Następnie zerujemy wszystkie macierze i rozpoczynamy obliczenia od początku wykorzystując nowo obliczony wektor zamiast wektora temperatur początkowych. Całość powinna działać w pętli aż do uzyskania czasu symulacji na poziomie zdefiniowanym przez użytkownika w pliku tekstowym.

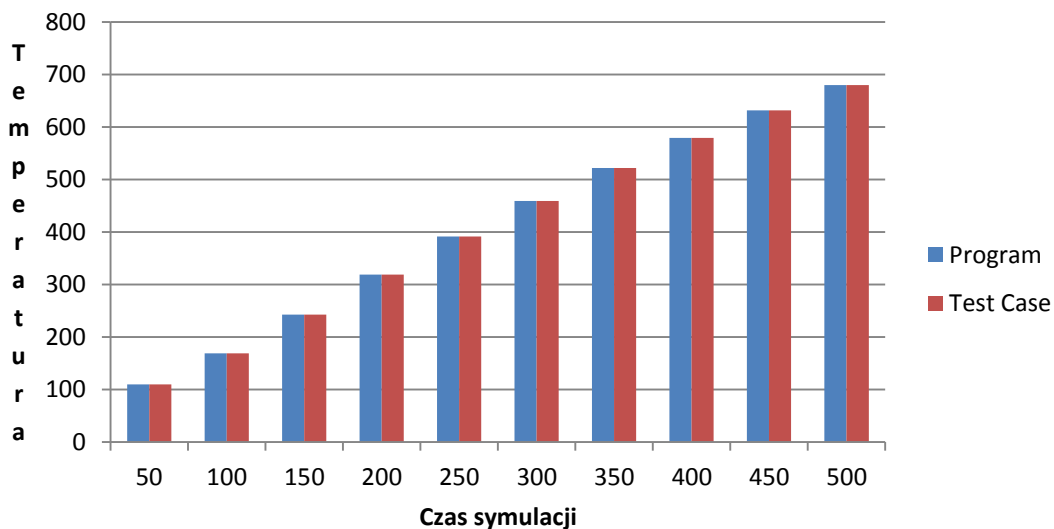
Testy Aplikacji

Test case 1

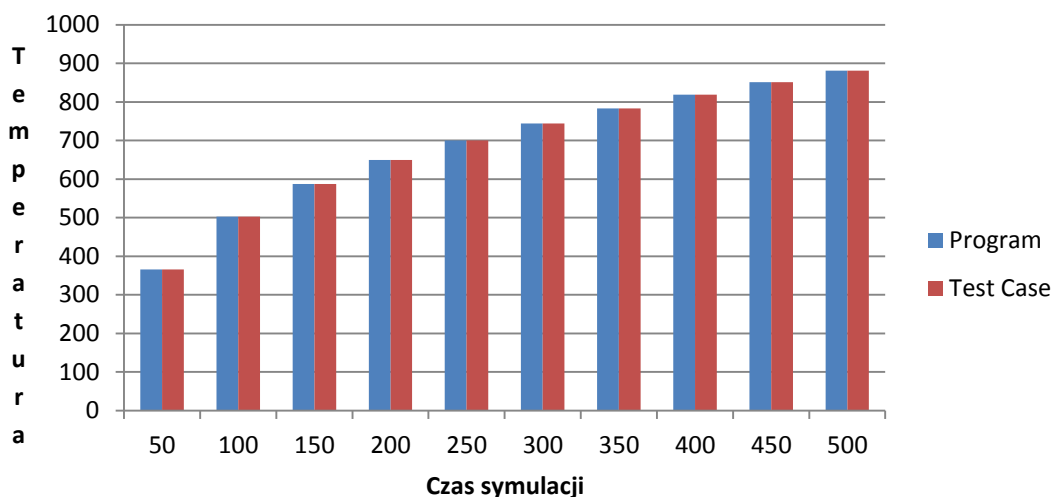
Wczytywane dane:

100 - initial temperature
500 - simulation time [s],
50 - simulation step time [s],
1200 - ambient temperature[C],
300 - alfa [W/m²K],
0.100 - H [m],
0.100 - B [m],
4 - N_H,
4 - N_B,
700 - specific heat [J/(kgC)],
25 - conductivity [W/(mC)],
7800 - density [kg/m³].

Porównanie temperatur minimalnych



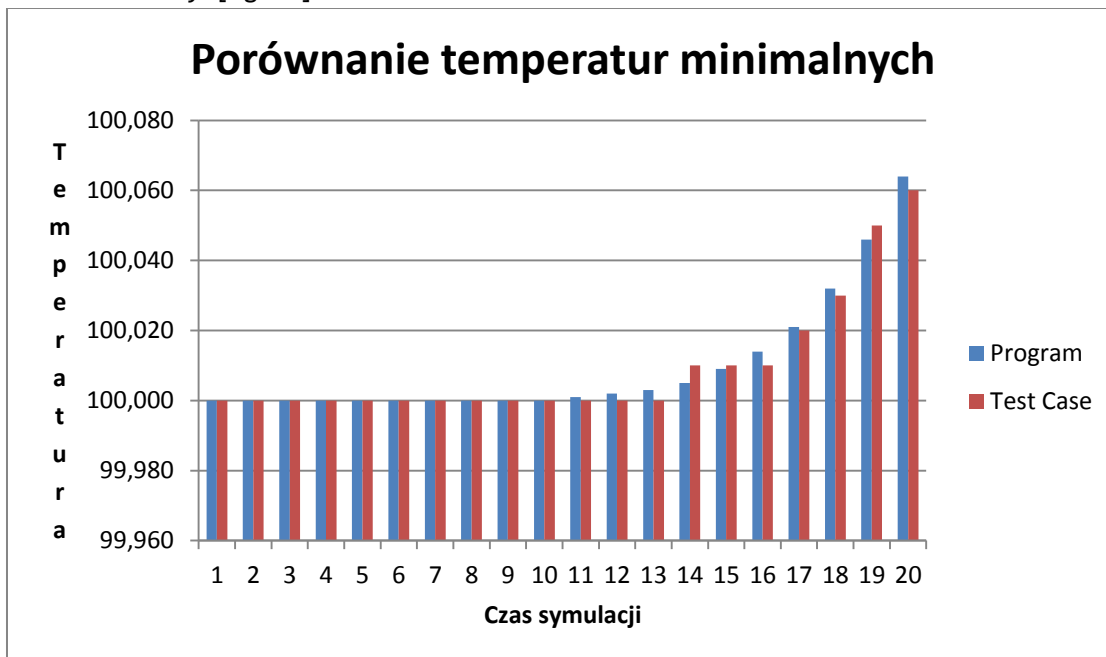
Porównanie temperatur maksymalnych



Test case 2

Wczytywane dane(czas skrócony ze względu na czas trwania obliczeń):

100 - initial temperature
 20 - simulation time [s],
 1 - simulation step time [s],
 1200 - ambient temperature[C],
 300 - alfa [W/m²K],
 0.100 - H [m],
 0.100 - B [m],
 31 - N_H,
 31 - N_B,
 700 - specific heat [J/(kg°C)],
 25 - conductivity [W/(m°C)],
 7800 - density [kg/m³].



Uwaga: rozbieżność w wykresie jest spowodowana nie zaokrągleniem wartości do 2giego miejsca po przecinku!

