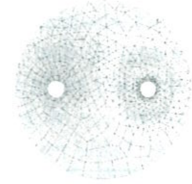
Wydawnictwo
Naukowe
AKAPIT

INFORMATYKA W TECHNOLOGII MATERIAŁÓW

Numer 4, Tom 4, Rok 2004



OBIEKTOWA REALIZACJA SOLWERA MES DLA PROCESU SPĘCZANIA

PAWEŁ J. MATUSZYK

OBJECT-ORIENTED REALIZATION OF THE FE SOLVER FOR COMPRESSION TEST

Abstract

The paper deals with developing and implementing object-oriented model of finite element solver for simulation of compression material tests. First, mathematical model of compression on the basis of incompressible nonlinear viscoplastic flow is considered. A weak form is developed using SUPG stabilization method. Several methods providing convergence of Newton-Raphson method for linearization of discretized model are presented. Then, purposefulness of applying of object-oriented technique is shortly discussed. Finally, description of object-oriented model of the FE solver is presented with short description of all classes.

1. MODEL FIZYCZNY I OBLICZENIOWY

1.1. Model konstytutywny

Ze względu na duże odkształcenia występujące w trakcie procesu spęczania zaniedbujemy odkształcenia sprężyste i stosujemy sztywno-lepkoplastyczny model konstytutywny w oparciu o równanie Nortona-Hoffa (Wagoner i Chenot, 2001):

$$\mathbf{s} = 2\eta\dot{\boldsymbol{\varepsilon}}, \tag{1}$$

gdzie $\dot{\boldsymbol{\varepsilon}} = \frac{1}{2}\left(\nabla\mathbf{u} + (\nabla\mathbf{u})^T\right)$ jest tensorem prędkości odkształcenia, $\mathbf{s}$ — dewiatorem naprężenia $\boldsymbol{\sigma} = \mathbf{s} - p\mathbf{I}$, gdzie $p = -\frac{1}{3}\sigma_{kk}$, a lepkość wyraża się wzorem (równanie Nortona-Hoffa):

$$\eta(\dot{\varepsilon}_i) = K\left(\sqrt{3}\dot{\varepsilon}_i\right)^{m-1} \qquad m \in [0,1]. \tag{2}$$

Zależności te umożliwiają wyrażenie tensora naprężenia $\boldsymbol{\sigma}$ na podstawie prędkości $\mathbf{u}$ oraz ciśnienia hydrostatycznego p.

Ze względu na brak odkształceń sprężystych dopuszczamy tylko odkształcenia postaciowe, a zatem materiał jest nieściśliwy. Implikuje to następujący warunek narzucany na pole prędkości:

$$\nabla \cdot \mathbf{u} = 0.$$

1.2. Tarcie

Do opisania tarcia występującego na powierzchni styku materiału i narzędzia stosujemy zależny od prędkości model Chena-Kobayashiego (1978) wyrażający naprężenie tarcia następująca zależnością:

$$\boldsymbol{\tau} = -mK\left[\frac{2}{\pi}\arctan\left(\frac{\mathbf{u}_s}{\alpha}\right)\right], \tag{3}$$

gdzie $\mathbf{u}_s$ jest prędkością względną pomiędzy materiałem i narzędziem, a stała α jest wielkością o $3-4$ rzędy mniejszą niż $\|\mathbf{u}_s\|$.

P.J. Matuszyk, Wydział Metalurgii i Inżynierii Materiałowej, Akademia Górniczo-Hutnicza w Krakowie

ISSN 1641-8581

1.3. Model procesu

Zastosowanie takiego modelu konstytutywnego prowadzi do tzw. *flow formulation* (Hartley, 1992). Model taki traktuje proces spęczania jako quasi-stacjonarny — w każdym kroku czasowym poszukiwane jest takie rozwiązanie $(\mathbf{u}, p)$, które spełnia warunek równowagi dla rozpatrywanej konfiguracji geometrycznej i fizycznej materiału oraz aktualnych warunków brzegowych. Odkształcenie materiału w czasie opisuje zależność:

$$\mathbf{x}^{n+1} = \mathbf{x}^n + h\left[\alpha \mathbf{u}^{n+1} + (1-\alpha)\mathbf{u}^n\right], \qquad (4)$$

gdzie $\mathbf{x}^i$ reprezentuje współrzędne opisujące geometrię odkształcanego materiału w i-tym kroku czasowym, a h jest wielkością kroku czasowego. Współczynnik $\alpha \in [0,1]$ określa żądany schemat całkowania po czasie.

1.4. Mocna forma

Mocna forma opisująca warunki równowagi dla materiału zajmującego obszar Ω ograniczony brzegiem Γ oraz zadanych warunków brzegowych dana jest następująco:

$$\begin{aligned}
\nabla \cdot \boldsymbol{\sigma} &= 0 && \text{na } \Omega \\
\nabla \cdot \mathbf{u} &= 0 && \text{na } \Omega \\
\mathbf{n} \cdot \boldsymbol{\sigma} &= \mathbf{t} && \text{na } \Gamma_N \\
\mathbf{u} &= \mathbf{u}_0 && \text{na } \Gamma_D,
\end{aligned}$$

gdzie $\mathbf{u}_0$ jest prędkością zadaną na brzegu Γ_D, a $\mathbf{t}$ siłą działającą na brzegu Γ_N. W rozpatrywanym przypadku, składowe styczne $\mathbf{t}$ będą równe tarciu $\boldsymbol{\tau}$, a składowa normalna $\mathbf{t}$ będzie równa poszukiwanej wartości siły, z jaką narzędzie działa na materiał. Na kinematyczne stopnie swobody odpowiadające składowej normalnej narzucone są warunki brzegowe Dirichleta ($\mathbf{u}_0$ równe prędkości narzędzia).

1.5. Sformułowanie Galerkina

Klasyczna metoda Galerkina prowadzi do słabej formy postaci: poszukujemy takiego $(\mathbf{u}, p) \in \mathbf{V} \times Q$, aby zachodziło:

$$\begin{cases}
\int\limits_\Omega \mathbf{s} : \dot{\boldsymbol{\varepsilon}}\, \mathrm{d}\Omega - \int\limits_\Omega p\, \nabla \cdot \mathbf{v}\, \mathrm{d}\Omega = \int\limits_{\Gamma_N} \mathbf{t} \cdot \mathbf{v}\, \mathrm{d}\Gamma_N & \forall \mathbf{v} \in \mathbf{V} \\
\int\limits_\Omega q\, \nabla \cdot \mathbf{u}\, \mathrm{d}\Omega = 0 & \forall q \in Q,
\end{cases}$$

gdzie $\mathbf{V} = \mathbf{H}^1(\Omega)$ oraz $Q = L^2(\Omega)$. Jest to klasyczne sformułowanie mieszane.

Aby istniało jednoznaczne rozwiązanie problemu dyskretnego $(\mathbf{u}_h, p_h) \in \mathbf{V}_h \times Q_h$, gdzie $\mathbf{V}_h \subset \mathbf{V}$, $Q_h \subset Q$, musi być spełniony dyskretny warunek Brezzi'ego-Babuški (1991). Kryterium to narzuca dość duże ograniczenia na przestrzenie aproksymujące $\mathbf{V}_h$ i Q_h, co w znaczny sposób zawęża ilość możliwych do zastosowania elementów skończonych.

1.6. Sformułowanie stabilizowane

W celu przezwyciężenia tych ograniczeń, zdecydowano się na zastosowanie metody stabilizacji sformułowania słabego metodą *Streamline Upwind Petrov-Galerkin* SUPG (Hughes i in., 1986, 1987). Metoda Petrova-Galerkina, z odpowiednio zmodyfikowanymi funkcjami wagowymi $\mathbf{w} = \mathbf{v} + \tau \nabla p$, których drugi człon wprowadza pewne zaburzenie na poziomie poszczególnych elementów K, prowadzi do następującej formy słabej:

$$\begin{cases}
\int\limits_\Omega \mathbf{s} : \dot{\boldsymbol{\varepsilon}}\, \mathrm{d}\Omega - \int\limits_\Omega p\, \nabla \cdot \mathbf{v}\, \mathrm{d}\Omega = \int\limits_{\Gamma_N} \mathbf{t} \cdot \mathbf{v}\, \mathrm{d}\Gamma_N \quad \forall \mathbf{v} \in \mathbf{V} \\
\int\limits_\Omega q\, \nabla \cdot \mathbf{u}\, \mathrm{d}\Omega + \sum\limits_{K \in \mathcal{T}_h} \int\limits_K \tau \nabla q \cdot \nabla p\, \mathrm{d}K \\
\qquad = \sum_{K \in \mathcal{T}_h} \int\limits_K \tau \nabla q \cdot (\nabla \cdot \mathbf{s})\, \mathrm{d}K \quad \forall q \in Q.
\end{cases} \qquad (5)$$

W tym przypadku, przestrzeń Q definiujemy następująco: $Q = H^1(\Omega)$. Współczynnik τ nazywany jest współczynnikiem stabilizacji. Za pracą Maniatty i in. (2001) przyjęto:

$$\tau = \frac{10^{-1} h_K^2}{2\eta} = \frac{10^{-1} h_K^2}{2K\left(\sqrt{3}\dot{\varepsilon}_i\right)^{m-1}}. \qquad (6)$$

W celu obliczenia prawej strony drugiego równania dla elementów wyższych rzędów (dla elementów liniowych jest ona równa 0) zastosowano metodę rekonstrukcji dewiatora naprężenia na poziomie poszczególnych elementów (Jansen i in., 1999).

1.7. Linearyzacja i dyskretyzacja

Układ równań (5) jest nieliniowy ze względu na prędkości $\mathbf{u}$. Do linearyzacji stosujemy metodę Newtona-Raphsona. Dla uproszczenia, nie będziemy linearyzować ostatniego członu drugiego równania (5) i będziemy go traktować jako dodatkową siłę działającą na układ. Po obliczeniu pochodnych oraz dyskretyzacji MES dostajemy następujący układ równań liniowych:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{S} \end{bmatrix} \cdot \begin{bmatrix} \delta\mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix}, \qquad \text{gdzie:} \qquad (7)$$

$$\begin{aligned}
\mathbf{A} &= \frac{\partial}{\partial \mathbf{u}} \left(\int\limits_\Omega \mathbf{s} : \dot{\boldsymbol{\varepsilon}}\, \mathrm{d}\Omega - \int\limits_{\Gamma_N} \boldsymbol{\tau} \cdot \mathbf{v}\, \mathrm{d}\Gamma_N \right) \\
\mathbf{B} &= -\int\limits_\Omega \mathbf{N}_p^T \mathbf{c}^T \mathbf{B}_u\, \mathrm{d}\Omega \\
\mathbf{f} &= -\frac{2m}{\sqrt{3}\pi} \int\limits_{\Gamma_N} \sigma_i \mathbf{N}_\Gamma^T \arctan\left(\frac{\mathbf{N}_\Gamma \mathbf{u}}{\alpha}\right) \mathrm{d}\Gamma_N \\
\mathbf{g} &= \int\limits_\Omega \mathbf{N}_p^T (\mathbf{c}^T \mathbf{B}_u \mathbf{u})\, \mathrm{d}\Omega - \sum_{K \in \mathcal{T}_h} \int\limits_K \tau \mathbf{B}_p^T \mathbf{B}_s \bar{\mathbf{s}}\, \mathrm{d}K.
\end{aligned}$$

$$\mathbf{S} = -\sum_{K\in\mathcal{T}_h}\int_K \tau \mathbf{B}_p^T\mathbf{B}_p \,\mathrm{d}K$$
$$-\sqrt{3}\int_\Omega \sigma_i\left(\sqrt{3}\dot{\varepsilon}_i\right)^{m-1}\mathbf{G}\mathbf{u}\,\mathrm{d}\Omega$$

Macierze $\mathbf{N}$ są macierzami funkcji kształtu, a $\mathbf{B}$ — macierzami pochodnych funkcji kształtu. Odpowiednie indeksy wskazują, dla jakich stopni swobody zdefiniowano powyższe macierze.

W celu zapewnienia lepszej zbieżności metody, zastosowano strategię progresywnej linearyzacji (Maniatty i in., 2001), polegającą na stopniowym (w kolejnych iteracjach) zwiększaniu nieliniowości modelu konstytutywnego, zaczynając od modelu liniowego (płynu newtonowskiego). Lepkość w równaniu konstytutywnym (1) wyrażamy następującą zależnością:

$$\eta = c_t \cdot K + (1-c_t)\cdot K(\sqrt{3}\dot{\varepsilon}_i)^{m-1}, \qquad (8)$$

gdzie $c_t = (1-t)e^{\beta t}$ oraz $0 \leqslant t \leqslant 1$. Dla $t=0$ dostajemy liniowy model lepkości, dla $t=1$ — silnie nieliniowy model docelowy, zaś dla wartości z przedziału $(0,1)$ — modele o charakterystykach pośrednich. Dodatkowy parametr β określa, jak szybko (w zależności od parametru t) model zbliża się do modelu docelowego.

W każdym kroku czasowym, obliczenie nowego rozwiązania $(\mathbf{u}, p)$ wymaga iteracyjnej procedury linearyzacji, na którą składa się:

- obliczenie współczynnika c_t (progresywna linearyzacja);
- rozwiązanie powyższego układu równań;
- procedura subinkrementacji: obliczenie nowego rozkładu prędkości $\mathbf{u}' = \mathbf{u} + \gamma\cdot\delta\mathbf{u}$, gdzie współczynnik $\gamma \in (0, 1.1]$ dobierany jest tak, aby minimalizował normę $|\mathbf{J}(\mathbf{u}')|$, gdzie funkcjonał $\mathbf{J}(\mathbf{u})$ oznacza residuum układu (5);
- zbadanie kryterium zbieżności $\frac{\|\delta\mathbf{u}\|}{\|\mathbf{u}\|} < 10^{-5}$.

1.8. Model termomechaniczny

Dla modelowania procesów termicznych zachodzących w spęczanej próbce stosujemy klasyczne równanie transportu ciepła następującej postaci:

$$\begin{cases} \nabla\cdot(k\nabla T) + \dot{w} = \rho c_p \dfrac{\partial T}{\partial t} \\ k(\mathbf{n}\cdot\nabla)T = \beta(T_{out} - T) + \dot{q} \\ T(x,y,z,0) = T_0(x,y,z). \end{cases}$$

Wielkości $\dot{w}$ oraz $\dot{q}$ definiują odpowiednio: ciepło generowane na skutek odkształcenia oraz powstające w wyniku tarcia i są zdefiniowane następującymi wzorami (Wagoner i Chenot, 2001):

$$\dot{w} = \nu\int_\Omega K\left(\sqrt{3}\dot{\varepsilon}_i\right)^{m+1}\mathrm{d}\Omega \qquad \nu\in[0.9, 1]$$
$$\dot{q} = \int_{\Gamma_N} mK|\mathbf{u}_s|\,\mathrm{d}\Gamma_N.$$

Po zastosowaniu metody Galerkina, a następnie dyskretyzacji dostajemy następujący układ równań:

$$\left(\mathbf{C} + \alpha h\mathbf{H}^{k+1}\right)\mathbf{T}^{k+1} = \qquad (9)$$
$$\left[\mathbf{C} - (1-\alpha)h\mathbf{H}^k\right]\mathbf{T}^k + h\left[(1-\alpha)\mathbf{P}^k + \alpha\mathbf{P}^{k+1}\right]$$

wiążący nowe pole temperatury $\mathbf{T}^{k+1}$ i stare $\mathbf{T}^k$. $\alpha\in[0,1]$ jest współczynnikiem określającym schemat całkowania po czasie.

1.9. Metody rozwiązywania układu równań liniowych

W rozpatrywanym przypadku, powstają dwa typy układów równań o macierzach rzadkich i symetrycznych. Oba typy macierzy różnią się jednak własnościami matematycznymi, co powoduje, iż do rozwiązania układów równań liniowych (URL) konieczne jest użycie różnych metod obliczeniowych:

- Problem mechaniczny: konieczne jest rozwiązanie układu równań (7), którego macierz posiada strukturę blokową: podmacierz $\mathbf{A}$ jest dodatnio określona, natomiast podmacierz $\mathbf{S}$ jest ujemnie półokreślona. Tak więc cała macierz jest nieokreślona. Do rozwiązania tego układu stosujemy metodę PMINRES (*Minimal Residual Method*) z blokowym uwarunkowaniem wstępnym metodą SSOR (symetrycznej nadrelaksacji) oraz metodą Jacobiego. Taką klasę preconditionerów zaproponowali Elman i in. (1996) dla układów równań powstających w wyniku dyskretyzacji równań Stokesa.
- Problem termomechaniczny: konieczne jest rozwiązanie układu równań (9), którego macierz jest dodatnio określona. W tym przypadku stosujemy do rozwiązania URL metodę PCG (sprzężonych gradientów) z uwarunkowaniem wstępnym metodą SSOR.

W obu przypadkach zastosowano do rozwiązania układów równań iteracyjne metody Kryłowa z uwagi na to, iż nie modyfikują macierzy układu, a jedynie potrzebują efektywnego algorytmu mnożenia macierz-wektor. Podstawową ideą metod Kryłowa jest wygenerowanie na podstawie macierzy układu $\mathbf{A}$ bazy przestrzeni afinicznej (przestrzeni Kryłowa):

$$\mathcal{K}_m(\mathbf{A}, \mathbf{x}_0) = \mathrm{span}\,\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{m-1}\mathbf{r}_0\},$$

gdzie $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$ oraz konstrukcja innej przestrzeni $\mathcal{L}$, zazwyczaj ściśle powiązanej z przestrzenią $\mathcal{K}$, a

następnie iteracyjne poszukiwanie nowego rozwiązania $\widetilde{\mathbf{x}} \in \mathbf{x}_0 + \mathcal{K}$ układu $\mathbf{A}\mathbf{x} = \mathbf{b}$ dla rozwiązania początkowego $\mathbf{x}_0$ według następującego algorytmu:

$$\widetilde{\mathbf{x}} = \mathbf{x}_0 + \delta, \quad \delta \in \mathcal{K}, \qquad (\mathbf{r}_0 - \mathbf{A}\delta, \mathbf{w}) = 0 \quad \forall \mathbf{w} \in \mathcal{L}.$$

W każdej zatem iteracji, dokonywana jest minimalizacja (ze względu na przestrzeń $\mathcal{L}$) uaktualnionego rozwiązania nad podprzestrzenią Kryłowa $\mathcal{K}$.

W metodzie PCG bierzemy $\mathcal{K} = \mathcal{L} = \mathcal{K}_m(\mathbf{A}, \mathbf{x}_0)$, zaś w metodzie PMINRES — $\mathcal{K} = \mathcal{K}_m(\mathbf{A}, \mathbf{x}_0)$ oraz $\mathcal{L} = \mathbf{A}\mathcal{K}$, co prowadzi do innych własności obu metod. Wspólną cechą metod PCG i PMINRES jest wykorzystanie krótkich rekurencji, na bazie algorytmu Lanczosa (co jest możliwe dzięki symetrii macierzy układu). Obie powyższe metody Kryłowa cechują się optymalnymi własnościami: metoda PCG minimalizuje normę energetyczną błędu, a metoda PMINRES — normę residualną.

Wybór takiego typu preconditionerów podyktowały względy praktyczne: zastosowanie metod Jacobiego i SSOR nie wymagało tworzenia dodatkowych struktur danych, wszystkie operacje uwarunkowania wstępnego działały „w miejscu" operując jedynie na oryginalnej macierzy układu.

2. DLACZEGO METODOLOGIA OBIEKTOWA?

W ostatnich latach pokazało się kilka prac, które wykazują przydatność metodologii obiektowej do tworzenia oprogramowania MES cechującego się dużą elastycznością, wydajnością i niezawodnością, np. Dubois-Pelerin (1992, 1993), Zimmermann i in. (1992).

Zasadniczym zagadnieniem analizy i projektowania obiektowego jest ujęcie rzeczywistości danego zagadnienia, które ma być docelowo zaimplementowane jako wykonywalny program komputerowy, w postaci pewnego zbioru obiektów, które zawierają powiązane ze sobą dane i serwisy operujące na tych danych. Obiekty te odwzorowują wybrane elementy rzeczywistości, ze szczególnym uwzględnieniem tych ich aspektów, które są istotne dla modelowanego zespołu zjawisk. Uporządkowane w pewną hierarchię, obiekty komunikują się wzajemnie pomiędzy sobą w celu rozwiązania danego zagadnienia.

Metodologia obiektowa umożliwia modelowanie w sposób bezpośredni, zgodny z heurystycznym podejściem analizy i opisu stosowanym przez człowieka, relacji zachodzących pomiędzy poszczególnymi składowymi danego systemu. Pozwala na abstrahowanie pewnych własności, cech wspólnych oraz różnic, uwzględnienie relacji zawierania oraz nade wszystko klasyfikację wszystkich składowych modelu, stosując formalną metodologię, którą można potem przełożyć, używając odpowiedniego języka obiektowego, do komputerowej implementacji danego systemu.

Centralnym pojęciem metodologii obiektowej jest klasa. Jest to wyodrębniony fragment rzeczywistości, charakteryzujący się swoistym zestawem cech: atrybutów (a więc składowych, stanowiących o istocie danej klasy) oraz metod (czyli zestawu czynności, które klasa może przejawiać, zarówno oddziałując ze światem zewnętrznym, jak i wewnątrz siebie).

Klasy te mogą tworzyć hierarchie — na poszczególnych poziomach, poczynając od najwyższego, odpowiadającego najwyższemu poziomowi abstrakcji, doprecyzowuje się funkcjonalność klas potomnych, wykorzystując przy tym własności odziedziczone od klas leżących wyżej w hierarchii (polimorfizm). Wyodrębnienie i połączenie poszczególnych danych i algorytmów oraz ukrycie wewnętrznych mechanizmów manipulacji dokonywanych na tych danych (enkapsulacja) powodują wzrost niezawodności kodu, łatwiejsze wykrywanie potencjalnych błędów oraz większą przenośność takiego oprogramowania.

Podstawowe składowe programu implementującego solwer MES, jak: siatka, węzły siatki, elementy, materiał, macierz, mogą być wprost opisane za pomocą metodologii obiektowej i zdefiniowane jako poszczególne klasy posiadające ściśle zdefiniowane struktury danych i stany (atrybuty klasy) oraz możliwe rodzaje zachowań (metody klasy).

3. MODEL OBIEKTOWY SOLWERA MES. HIERARCHIA KALS

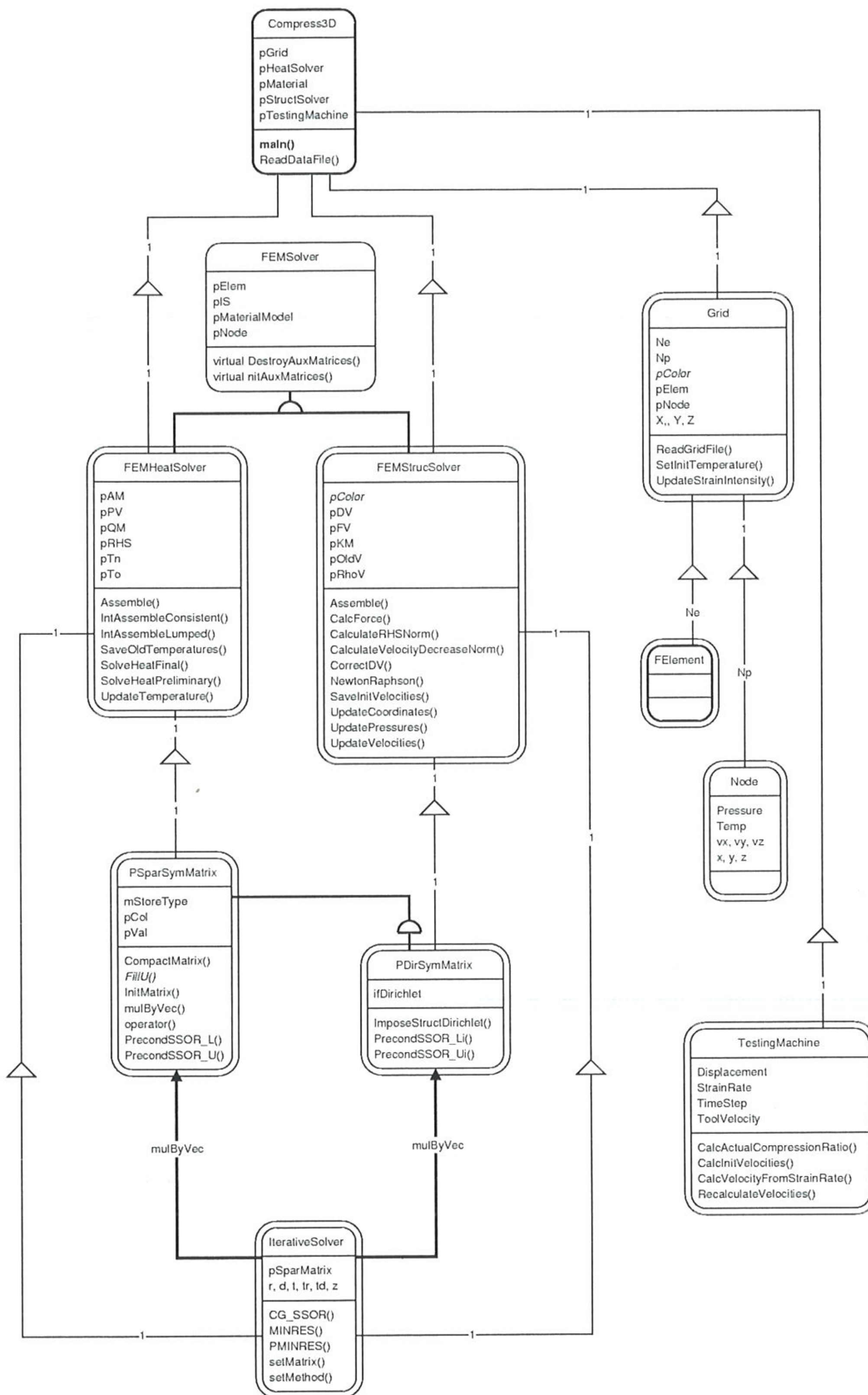
Przedstawiamy zaproponowany model obiektowy programu, który odwzorowuje proces obliczeniowy spęczania próbek metodą elementów skończonych. Schemat obiektowy całego programu implementującego solwer MES, bez szczegółowego przedstawienia hierarchii klas implementujących elementy skończone, przedstawiono na rysunku 1. Poniżej omówimy pokrótce poszczególne składowe modelu z omówieniem ich wewnętrznej struktury i funkcjonalności.

3.1. Klasa `Grid`

Klasa `Grid` służy do przechowywania siatki elementów skończonych oraz manipulowania nią. Na siatkę elementów skończonych składają się informacje o węzłach oraz elementach. Potrzebne są zatem dwie klasy, które opiszą węzeł (klasa `Node`) oraz element skończony hierarchia klas `FElement`.

Do podstawowych atrybutów klasy `Grid` należą:

- tablica `pNode`, której elementami będą obiekty klasy `Node`, służące do przechowywania współrzędnych oraz stopni swobody węzłów siatki wraz z rozmiarem tej tablicy (czyli ilością węzłów siatki) `Np`;
- tablica wskaźników `pElem` do obiektów potomnych klasy `FElement` wraz z jej rozmiarem (czyli ilością elementów siatki) `Ne`;
- `X`, `Y`, `Z` do przechowywania początkowych wymiarów fizycznych próbki (siatki).



Rysunek 1: Schemat obiektowy solwera MES. (*Object diagram of FE solver.*)

Do właściwej interpretacji operacji wykonywanych na poziomie klasy Grid przechowujemy w tej klasie w atrybutach mProblemType, mGrid, mElemType informacje o typie rozwiązywanego problemu, siatki oraz użytych elementów skończonych. Warto zaznaczyć, iż jeśli chodzi o typ elementów danej siatki, to pod tym względem siatka jest zawsze jednorodna — używamy tylko jednego typu elementu skończonego dla wszystkich elementów siatki.

Do wczytania siatki z pliku służy metoda ReadGridFile(). Dokonuje ona parsingu pliku, inicjuje tablice do przechowywania danych oraz wywołuje konstruktory odpowiednich klas w celu utworzenia węzłów i elementów siatki.

Pozostałe metody służą do wykonywania globalnych operacji na siatce, jakie mają miejsce pod koniec każdego kroku czasowego algorytmu spęczania. Są to metody UpdateCoordinates() oraz UpdateStrainIntensity(). Obie, wymagając jedynie podania wielkości kroku czasowego, bazują na informacjach przechowywanych w samych elementach skończonych.

W programie tworzona jest tylko jedna instancja klasy Grid. Udostępnia ona wskaźniki do tablic pNode oraz pElem obiektom wykonującym obliczenia MES.

3.2 KLASA FElement ORAZ KLASY POCHODNE

Klasa FElement stanowi jednolity interfejs definiujący dla wszystkich klas potomnych różne typy elementów skończonych, użytych w programie. Stanowią go metody, które są wywoływane przez zewnętrzne obiekty innych klas, które potrzebują wyników różnorodnych operacji wykonywanych na konkretnych obiektach skończonych. Te, i tylko te funkcje są dostępne z zewnątrz. Użytkownika tej klasy interesuje np. tylko układ równań zbudowany dla danego elementu skończonego, nie wnika on, w jaki sposób jest on budowany. To, jak jest to faktycznie wykonywane dla danego elementu i jego typu jest zaszyte i niewidoczne z zewnątrz w poszczególnych klasach specyfikujących dany element skończony. Mamy tu zatem do czynienia z enkapsulacją danych, czyli oddzieleniem użytkownika zewnętrznego od wewnętrznej specyfiki danego obiektu, umożliwiając mu jedynie wykonywanie pewnych, ściśle określonych, operacji na tym obiekcie. Wszystkie metody klasy FElement są funkcjami wirtualnymi, tzn. zewnętrzny użytkownik wywołując te funkcje nie będzie specyfikował wprost, na rzecz jakiego typu obiektu zostanie wywołana dana funkcja, a przez to interpretował, jaka konkretna metoda i w jakiej klasie będzie wywołana — czynność ta będzie wykonana automatycznie w czasie działania programu, po sprawdzeniu typu obiektu, na rzecz którego wywołujemy funkcję wirtualną. Użycie funkcji wirtualnych stanowi jedno z większych dobrodziejstw, jakie niesie ze sobą programowanie obiektowe — zapewnia spełnienie postulatu zachowania abstrakcyjnego modelu danych i operowania nim na różnych poziomach abstrakcji, co umożliwia bardziej elastyczne wykorzystanie takiego modelu oraz jego modułową budowę. Niestety, powoduje to pewien dodatkowy narzut czasowy. Jak jednak wykażemy później, dla przypadku implementowanego solwera MES nie stanowi to o istotnym, względnie zauważalnym, pogorszeniu wydajności programu. Narzut ten jest znikomy w porównaniu z czasem rozwiązania URL.

Do opisu konkretnego elementu skończonego, który wykorzystamy w solwerze MES potrzebne są następujące dane: indeksy węzłów elementu (tablica pNode), indeksy elementów sąsiednich (atrybut pNeig), indeksy rodzajów warunków brzegowych, o ile takie występują, na poszczególnych ścianach elementu (informacje te przechowuje się także w tablicy pNeig, ponieważ występowanie warunku brzegowego wyklucza istnienie sąsiada wzdłuż danej ściany elementu — w ten sposób oszczędzamy miejsce w pamięci), punkty całkowania Gaussa (GaussP), wartości intensywności prędkości odkształcenia w punktach Gaussa (pESR), wartości intensywności odkształcenia w punktach Gaussa (pStrainIntensity), wartości intensywności naprężenia w punktach Gaussa (pStressIntensity).

Ponadto, każdy element skończony powinien mieć dostęp do: wskaźnika do tablicy węzłów siatki oraz wskaźników do funkcji w klasie MaterialModel służących do obliczania wartości parametrów fizycznych materiału.

Podstawowe metody klasy FElement to:

- BuilStiffnessMatrix() — funkcja ta buduje macierz sztywności oraz wektor wyrazów wolnych dla części mechanicznej procesu;
- BuilCapacityMatrix() — funkcja ta buduje macierz pojemności, macierz przewodnictwa i wektor wyrazów wolnych dla części termomechanicznej procesu;
- CalcRHS() — funkcja oblicza wektor będący prawą stroną zlinearyzowanego układu równań; norma tego wektora jest kontrolowana w czasie iteracyjnej procedury Newtona-Raphsona;
- CalcForce() — funkcja oblicza składowe siły działającej na tej części brzegu elementu, która stanowi część brzegu obszaru stykającego się z narzędziem;
- UpdateStrainIntensity() — funkcja dokonuje obliczenia dla każdego punktu całkowania nowych wartości intensywności odkształcenia w kolejnym kroku czasowym zgodnie z równaniem $\varepsilon_i^{new} = \varepsilon_i^{old} + \dot{\varepsilon}_i \cdot \delta t$;
- IsDirchletStruct() — funkcja sprawdza, czy na brzegu danego elementu zadany jest warunek brzegowy pierwszego rodzaju, a jeśli tak, to jego typ oraz indeksy węzłów, które należą do takiego brzegu.

SPECYFIKACJA KLAS IMPLEMENTUJĄCYCH ELEMENTY SKOŃCZONE.

Jak już wspomniano na początku tego rozdziału, zaimplementowane elementy możemy ogólnie podzielić na dwie grupy: elementy dwu- i trójwymiarowe. Różnią się one przede wszystkim: liczbą stopni swobody na węzeł oraz specyfiką brzegu, na którym występują warunki brzegowe (2D — krzywa, 3D — powierzchnia). Grupę elementów dwuwymiarowych podzielić można dalej na: elementy dla płaskiego stanu odkształcenia (PSC) oraz elementy dla osiowosymetrycznego stanu odkształcenia (AXI). Różnice pomiędzy nimi odzwierciedlone są tylko w nieco odmiennej budowie macierzy — pojawia się dodatkowa składowa tensora prędkości odkształcenia; odmienną postać ma także jakobian (choć współczynniki potrzebne do jego obliczenia są identyczne). Można jednak wyodrębnić dla wszystkich klas implementujących elementy dwuwymiarowe pewne wspólne metody i atrybuty, które zaimplementowane są w klasach zbiorczych, zdefiniowanych dla poszczególnych typów elementów. Do wspólnych składowych należą:

- współczynniki jakobianu — zarówno dla PSC, jak i AXI są one identyczne,
- współrzędne i wagi punktów Gaussa,
- metoda `IsDirichletStruct()` oraz `UpdateStrainIntensity()`,
- metody prywatne: obliczająca współczynniki jakobianu (`CalcJacobian()`, `CalcFrictionJacobCoeff()`), wartości funkcji kształtu w danym punkcie (`CalcN()`), sprawdzające występowanie na boku elementu warunku brzegowego (`isFriction()`, `isConvection()`), obliczająca średnicę elementu (`CalcElemDiam()`) i budujące macierze termomechaniczne w wybranym punkcie Gaussa (`CalcTempH()` oraz `CalcTempCandP()`).

Do metod, których implementacja jest ściśle uzależniona od typu elementu skończonego, należą:

- metody budujące kompletny układ równań na poziomie elementu (`BuildCapacityMatrix()` i `BuildStiffnessMatrix()`) oraz wektor prawej strony (`CalcRHS()`);
- metoda `CalcB()` — obliczająca wartości pochodnych przestrzennych;
- metody `CalcTempHBound()` i `CalcTempPBound` obliczające wkład do elementowego URL od warunków brzegowych;
- metoda `CalcDerMatrices()` obliczająca macierze i wielkości pomocnicze w danym punkcie całkowania; metoda ta oblicza bądź wszystkie wymienione wielkości, bądź tylko wybrane, w zależności od potrzeby zewnętrznej funkcji, która tę metodę wywołuje.

Implementując wymienione powyżej metody skupiono się przede wszystkim nad ich efektywnością. Z jednej strony starano się ująć strukturę opisanych klas w sposób jak najbardziej jednolity, wyabstrahowując najbardziej istotne schematy użycia w kilka ogólnych i uniwersalnych metod, z drugiej zaś strony — wykorzystując mechanizmy obiektowe udostępnione przez język C++ — dostosować i efektywnie zaimplementować dla każdej z poszczególnych klas reprezentujących typ elementu skończonego.

Większość operacji wykonywana jest na statycznych składowych klasy `FElement`, przez co zminimalizowano użycie lokalnych zmiennych w każdej z metod, a dzięki temu zmniejszono narzut czasowy związany z inicjalizacją zmiennych lokalnych funkcji na stosie.

Wykorzystano fakt, iż wszystkie macierze, tak w sformułowaniu mechanicznym, jak i termomechanicznym, są macierzami symetrycznymi. Najbardziej kosztowne obliczeniowo operacje całkowania numerycznego (pętla zewnętrzna: po kolejnych punktach Gaussa) zostały wykonane tylko dla części trójkątnej górnej macierzy. Po jej ostatecznym wyliczeniu kopiowano wyniki do części trójkątnej górnej, a następnie (jeśli zaszła taka potrzeba) modyfikowano je, uwzględniając warunki brzegowe pierwszego rodzaju.

ZAIMPLEMENTOWANE TYPY ELEMENTÓW SKOŃCZONYCH.

W rozpatrywanym solwerze zaimplementowano następujące typy elementów skończonych, z ciągłym polem ciśnień:

- 2D P1/P1: klasa `E2D_T3eq_PSC` dla płaskiego stanu odkształcenia oraz klasa `E2D_T3eq_AXI` dla osiowosymetrycznego stanu odkształcenia,
- 2D Q1/Q1: klasa `E2D_Q4eq_PSC` dla płaskiego stanu odkształcenia oraz klasa `E2D_Q4eq_AXI` dla osiowosymetrycznego stanu odkształcenia,
- 2D Q2/Q1: klasa `E2D_Q9Q4_PSC` dla płaskiego stanu odkształcenia oraz klasa `E2D_Q9Q4_AXI` dla osiowosymetrycznego stanu odkształcenia,
- 3D Q1/Q1: klasa `E3D_H8eq` dla trójosiowego stanu odkształcenia.

Główne różnice stanowiące o odmienności tych klas można podzielić na ilościowe i jakościowe. Do tych pierwszych można zaliczyć przede wszystkim ilość węzłów w elemencie oraz ilość punktów całkowania, co jest ściśle powiązane ze stopniem interpolacji. Do różnic jakościowych zaliczymy przede wszystkim odmienne budowy macierzy pochodnych funkcji kształtu **B**, co jest spowodowane tym, iż tensory prędkości odkształcenia dla płaskiego stanu odkształcenia (PSC), osiowosymetrycznego stanu odkształcenia (AXI) oraz trójosiowego stanu odkształcenia (3D)

są zasadniczo różne. Ujednolicony dla wszystkich klas schemat algorytmów budowy kolejnych macierzy i wektorów oraz jednolity interfejs sprawiają, że istotne modyfikację dotyczą tylko niewielkich partii kodu. Dzięki temu łatwiej jest wychwycić ewentualne błędy, które mogą pojawić się przy implementowaniu kolejnych metod dla poszczególnych klas reprezentujących elementy skończone.

Schemat obiektowy rodziny klas `FElement` przedstawiono na rysunku 2.

3.3. Klasa `TestingMachine`

Do opisu maszyny testowej stworzono klasę `Testing Machine`. Ma ona za zadanie:

- przechowywać informacje dotyczące zadanych parametrów odkształcenia (stała prędkość narzędzia czy stała prędkość odkształcenia, krok czasowy, wielkość odkształcenia) — atrybuty: `mMachineType`, `mProblemType`, `ToolVelocity`, `StrainRate`, `TimeStep`, `Displacement`, `CompressionRatio`;
- w zależności od potrzeby, obliczać prędkość narzędzia w danym kroku czasowym — metody: `CalcVelocityFromStrainRate()`, `RecalculateVelocities()`;
- w chwili uruchomienia procesu, dla zadanych warunków brzegowych oraz geometrii próbki, wygenerować pole prędkości początkowych całego obszaru próbki — metoda `CalcInitVelocities()`;
- kontrolować proces odkształcenia (kryterium stopu dla symulowanej próby plastometrycznej) — metoda `CalcActualCompressionRatio`.

W programie używana będzie tylko jedna instancja tej klasy, ponieważ zakłada się, że w trakcie całego procesu, nie ulegają zmianie warunki testu.

Ponieważ w każdym kroku czasowym dokonywana jest linearyzacja równań Stokesa metodą Newtona-Raphsona, w celu osiągnięcia lepszej (a niekiedy w ogóle) zbieżności, należy znaleźć przybliżenie rozwiązania i użyć go jako rozwiązania początkowego do procedury linearyzacji. Do wygenerowania początkowego pola prędkości można stosować różne metody. W projektowanym programie zaimplementowano metodę, która jest najmniej złożona obliczeniowo oraz daje zadowalające przybliżenie początkowe. Generowane jest pole prędkości początkowych z geometrii siatki oraz warunków brzegowych. W praktyce, najpierw znajdujemy pole prędkości uwzględniając narzucone warunki Dirichleta na brzegach siatki, a następnie generujemy prędkości wewnątrz siatki stosując rozkład liniowy (w danym kierunku prędkość zmienia się liniowo od jednego brzegu do drugiego).

3.4. Klasa `MaterialModel`

Do opisu właściwości fizycznych materiału stworzono klasę `MaterialModel`. Ma ona za zadanie przechowywać i opisywać modele wszystkich istotnych wielkości fizycznych oraz umożliwiać ich obliczenie dla danych warunków procesu. Każdy model wielkości fizycznej jest przechowywany jako:

- zadany model matematyczny (równanie), co w klasie przekłada się na metodę obliczającą zadaną wielkość fizyczną — `K*Model*()`, `CpModel*()`, `RhoModel*()`, `SiModel*()` oraz
- współczynniki tego modelu, przechowywane jako atrybut klasy (tablica) — `k*_coeff`, `cp_coeff`, `rh_coeff`, `si_coeff`.

W opisany powyżej sposób przechowywane są modele dla: współczynników przewodnictwa cieplnego (k_x, k_y, k_z), ciepła właściwego (c_p), gęstości (ρ) oraz modelu konstytutywnego (σ_i). Oprócz tego, w klasie przechowujemy parametry skalarne, takie jak: współczynniki konwekcji (`ToolConvect`, `AmbiConvect`), temperatury ciała (`BodyTemp`), narzędzia (`ToolTemp`) i otoczenia (`AmbiTemp`), modelu tarcia (`FricCoeff i AlphaCoeff`) oraz współczynnik wrażliwości dla modelu lepkoplastycznego (`StrRateSensFact`).

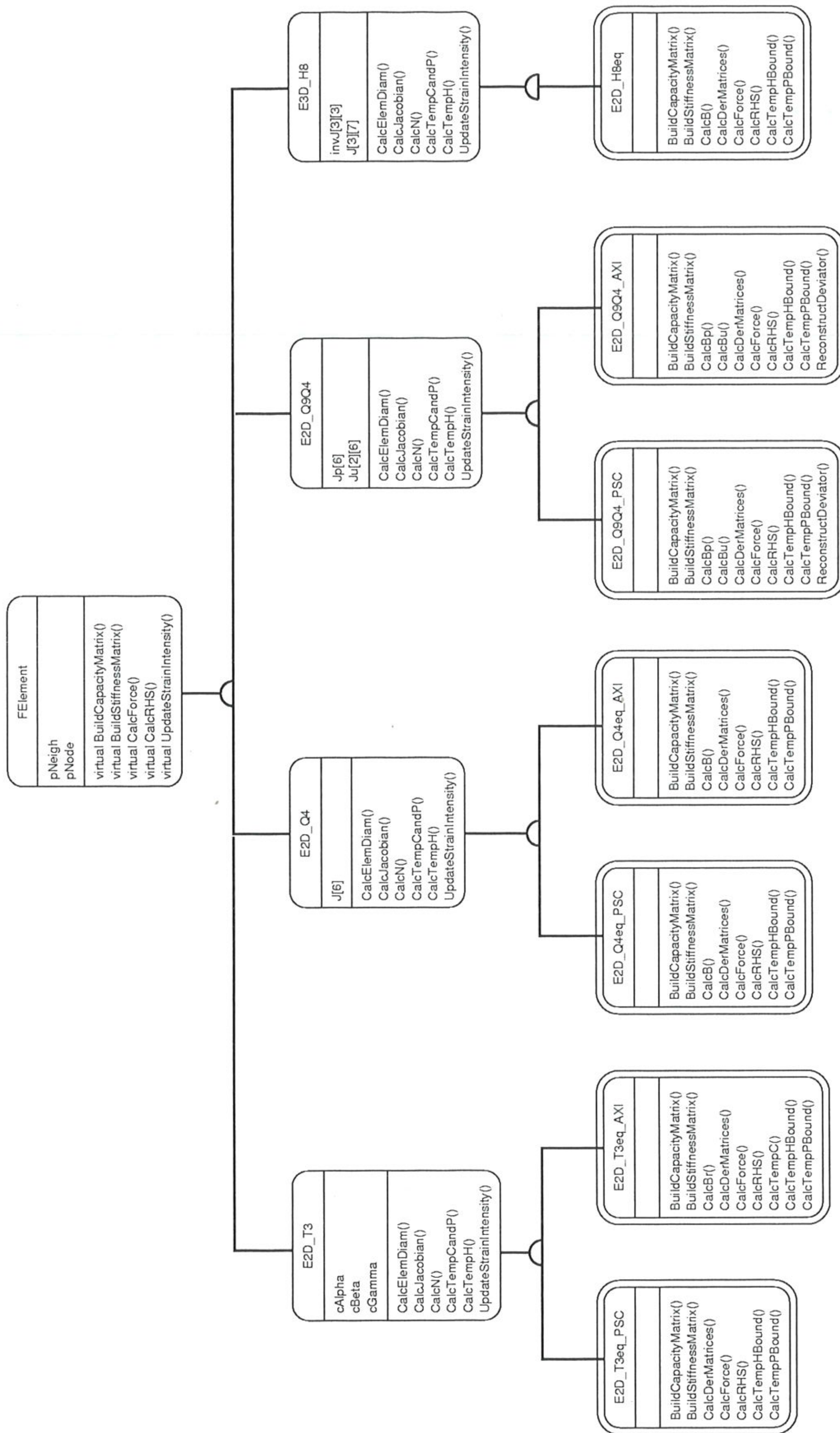
Klasa ta używana jest jako kontener różnych modeli fizycznych. Po ustawieniu adekwatną metodą konkretnego modelu oraz jego współczynników, klasa ta udostępnia na zewnątrz wskaźnik do funkcji pozwalającej obliczyć wartość danej zmiennej dla tego modelu oraz zespołu parametrów zewnętrznych dostarczonych do funkcji.

Ze względu na wydajność (wspomniane funkcje obliczające wartości parametrów fizycznych są wywoływane dla każdego punktu Gausa, w każdym elemencie), nie zdecydowano się na wywoływanie tych funkcji wprost, jako metod działających na obiekcie klasy `MaterialModel`, ale udostępniono wskaźniki do tych funkcji klasom implementującym elementy skończone — stanowią dla nich składowe statyczne, czyli wspólne dla wszystkich elementów skończonych.

3.5. Klasy implementujące Solwery MES

Do realizacji głównych obliczeń MES dla problemu odkształcania ciała oraz opisu jego termodynamiki zaimplementowno dwie klasy: `FEMStructSolver` i `FEMHeatSolver`. Dla obu klas utworzono klasę nadrzędną `FEMSolver`, która implementuje wspólne dla nich atrybuty i metody:

- wskaźniki do tablic zawierających indeksy węzłów (`pNode`) i elementów siatki (`pElem`),
- wskaźnik `pMaterialModel` do klasy opisującej model materiałowy próbki,



Rysunek 2: Schemat rodziny klas implementujących elementy skończone. (*Class hierarchy diagram of FE family.*)

- wskaźnik pIS do klasy IterativeSolver implementującej solwer iteracyjny do rozwiązywania układów równań liniowych.

Ponadto, w klasie tej zadeklarowano chronione wirtualne funkcje do tworzenia i usuwania pomocniczych tablic, które są wykorzystywane m.in. w procesie budowy globalnego URL.

3.5.1. *Klasa* FEMHeatSolver

Zadaniem tej klasy jest implementacja solwera opisującego termiczne zjawiska zachodzące w odkształcanym materiale. Do najważniejszych zadań tej klasy należą:

- budowa w każdym kroku czasowym układu (9),
- rozwiązanie tego układu równań,
- zapisanie obliczonego rozkładu temperatury do struktur opisujących węzły siatki.

Klasa FEMHeatSolver posiada następujące atrybuty:

- pAM i pQM — wskaźniki do obiektów klasy PSpar SymMatrix, implementujących symetryczne macierze rzadkie,
- pTo i pTn — wektory reprezentujące stare i nowe pole temperatur,
- pRHS i pPV — wektory służące do obliczania prawej strony URL,
- hState — określa, który z dwóch pośrednich kroków obliczeń będzie wykonywany w danej chwili (krok wstępny HEAT_PRELIMINARY i krok końcowy HEAT_FINAL),
- hType — określa, jak budowana jest macierz masowa **C** (może ona być budowana w sposób standardowy HEAT_CONSISTENT lub w postaci tzw. *lumped* (Wagoner i Chenot, 2001) HEAT_LUMPED),
- Alpha i TimeStep — określają schemat całkowania po czasie oraz krok czasowy.

Ze względu na to, iż parametry σ_i, k, β, ρ, c_p, $\dot{w}$ i $\dot{q}$ nie są stałe, ale same zależą od temperatury, rozpatrywany układ równań jest układem nieliniowym. Mając to na uwadze, przyjęto uproszczoną strategię rozwiązania tego układu równań. W każdym kroku czasowym, procedurę obliczenia nowego rozkładu temperatury rozbito na dwa kroki pośrednie. W każdym z nich, ustalamy wartości parametrów fizycznych i rozwiązujemy liniowy układ równań.

Do realizacji algorytmu budowy URL zaimplementowano w klasie FEMHeatSolver następujące metody:

- Assemble() — metoda ta, używając prywatnych metod IntAssembleConsistent() oraz IntAssembleLumped(), w zależności od rodzaju kroku (określonego atrybutem nState) oraz typu macierzy masowej (określonego atrybutem nType) buduje globalny układ równań;
- SolveHeatPreliminary() — rozwiązuje URL obliczając pole temperatur $\mathbf{T}_{int}$;
- SolveHeatFinal() — rozwiązuje układ równań obliczając pole temperatur $\mathbf{T}_{new}$;
- UpdateTemperature() — zapisuje nowo obliczone temperatury do struktur opisujących węzły siatki.

3.5.2. *Klasa* FEMStructSolver

Zadaniem tej klasy jest implementacja solwera obliczającego nowe pole prędkości i nowe pole ciśnień. Podstawowym zadaniem tej klasy jest linearyzacja układu równań (5) metodą Newtona-Raphsona, obliczanie siły działającej na odkształcane ciało oraz aktualizacja danych opisujących węzły siatki nowymi wartościami prędkości i ciśnień.

Klasa FEMStructSolver posiada następujące atrybuty:

- Ns, Nv, Np — określające liczbę: wszystkich stopni swobody oraz stopni swobody związanych z prędkościami i ciśnieniami;
- pKM — wskaźnik do obiektu klasy PDirSymMatrix, implementującego macierz rzadką, służącego do przechowywania macierzy sztywności;
- pDM— wskaźnik do obiektu klasy ForceDirMatrix, implementującego macierz rzadką, służącego do przechowywania tych wierszy macierzy sztywności, w których zadano warunki brzegowe Dirichleta związane z tarciem; macierz ta służy do obliczenia siły działającej na odkształcane ciało;
- pFV i pDV — wektor prawej strony i wektor rozwiązań;
- pOldV i pRhoV — pomocnicze wektory do zapisu starego pola prędkości oraz obliczania normy przyrostu prędkości;

Procedura linearyzacji. Główną metodą klasy jest funkcja NewtonRaphson(). Wymaga ona podania dwóch parametrów określających dokładność, z jaką rozwiązujemy sam układ równań liniowych (URL) oraz z jaką dokonujemy linearyzacji ϵ (jest to maksymalna norma ρ_{RHS} i norma ρ_u). Metoda ta implementuje procedurę linearyzacji, który przedstawiamy bliżej jako poniższy algorytm:

- **repeat**
  - oblicz współczynnik c_t;
  - buduj URL dla $(\mathbf{u}_{old}, \mathbf{p}_{old})$;
  - rozwiąż URL $\Rightarrow (\delta\mathbf{u}, \mathbf{p}_{new})$;
  - $\mathbf{u}_{old} = \mathbf{u}$, aktualizuj pole ciśnień wartościami $\mathbf{p}_{new}$;
  - minimalizacja $\rho_i^{RHS} = \|RHS(\mathbf{u})\|$ — przeszukiwanie liniowe;

- minimalizacja $\rho_i^{RHS} = \|RHS(\mathbf{u})\|$ — minimalizacja kwadratowa;
- $\rho^u = \|\frac{\delta\mathbf{u}}{\mathbf{u}}\|$;

until $\rho^u < \epsilon \wedge \rho^{RHS} < \epsilon$

Najistotniejszą jego częścią, oprócz rzecz jasna budowy i rozwiązania samego URL, jest procedura subinkrementacji, która decyduje szybkości zbieżności całej procedury linearyzacji. W celu przyspieszenia, a niekiedy zapewnienia zbieżności metody Newtona-Raphsona, zaimplementowano mechanizm progresywnej linearyzacji modelu oraz procedurę subinkrementacji, która składa się z dwóch etapów:

1. Minimalizacja globalna — przeszukiwanie całego przedziału dla różnych wartości współczynnika przyspieszenia γ: począwszy od wartości 1.1 poprzez kolejne, malejące w ciągu geometrycznym (iloraz równy α). Ich liczba zależy od tego, czy jest to pierwsza (i wtedy jest ich więcej), czy też kolejna iteracja procedury linearyzacji. Spośród tych wartości wybierana jest ta, dla której norma $\|\rho^{RHS}\|$ jest minimalna i wraz z dwoma sąsiednimi wartościami stanowi dane wejściowe dla drugiego kroku minimalizacji.
2. Minimalizacja lokalna — metodą parabol (`MinQuad`). W kolejnych iteracjach, przez trzy sąsiednie punkty — pary (γ_i, ρ_i^{RHS}) — prowadzimy parabolę, obliczamy nową wartość γ, która minimalizuje tę funkcję kwadratową i tworzymy nową trójkę par, która zawiera w sobie poszukiwane minimum.

Pozostałe procedury pomocnicze zaimplementowane w klasie `FEMStructSolver`, używane w procesie linearyzacji, to:

- `Assemble()` — metoda buduje globalny układ równań liniowych dla każdej iteracji metody Newtona-Raphsona wprowadzając warunki brzegowe Dirichleta.
- `MinQuad()` — metoda oblicza wartość współczynnika γ minimalizującego normę residualną.
- `CalcVelocityDecreaseNorm()` — metoda oblicza normę $\|\frac{\delta\mathbf{u}}{\mathbf{u}}\|$.
- `CalculateRHSNorm()` — metoda oblicza normę residualną dla pola prędkości.
- `CalcForce()` — obliczająca siłę, z jaką, w danym kroku czasowym, narzędzie działa na materiał.

3.6. Klasa `IterativeSolver`

Klasa `IterativeSolver` implementuje algorytmy rozwiązywania układów równań linowych. Są to metody PCG oraz PMINRES. Ponieważ obie zaimplementowane metody iteracyjne opierają swoje działanie na generowaniu nowych rozwiązań w oparciu o krótkie rekurencje, do ich działania potrzebnych jest kilka pomocniczych wektorów: dla metody PCG — 4 wektory, dla PMINRES — 6 wektorów. Ich rozmiar jest równy rozmiarowi problemu (ilości stopni swobody siatki). Ponieważ, jeśli nie mamy do czynienia z *remeshingiem* siatki, ilość węzłów siatki oraz jej topologia pozostają bez zmian w kolejnych iteracjach linearyzacji oraz w kolejnych krokach czasowych, wektory te wystarczy utworzyć tylko jeden raz, przy powołaniu do życia instancji klasy `Iterative Solver`. Jeśliby w trakcie działania solwera MES zaszła potrzeba, wtedy wektory te można by było zlikwidować i utworzyć nowe, o odpowiednich rozmiarach. Zysk z takiego rozwiązania jest oczywisty: nie musimy przy każdym wywołaniu metody rozwiązującej URL tworzyć na nowo, a przy jej opuszczeniu likwidować tych pomocniczych struktur danych — jest to oczywista oszczędność czasu, który w przeciwnym wypadku musiałby być poświęcony na wykonanie tych dodatkowych operacji.

Podstawową funkcją klasy `IterativeSolver` jest metoda `Solve()` rozwiązująca liniowy układ równań. Funkcja wywołuje poprzez wskaźnik odpowiednią metodę, implementującą jedną z metod iteracyjnych. Aby funkcja `Solve()` mogła działać poprawnie, konieczne jest uprzednie ustawienie odpowiednich atrybutów klasy:

- `SetProblemSize()` — określenie rozmiaru problemu, równego rozmiarowi URL,
- `SetMatrix()` — ustawienie wskaźnika na zewnętrzny obiekt przechowujący macierz URL (jest to obiekt klasy `PSparSymMatrix`),
- `SetMethod()` — ustawienie rodzaju metody iteracyjnej rozwiązującej URL; funkcja ta ustawia wskaźnik zdefiniowany wewnątrz klasy na konkretną funkcję implementującą daną metodę iteracyjną.

Ponadto klasa udostępnia funkcje, które po zakończeniu procesu rozwiązywania URL danych o jakości rozwiązania oraz samej metody:

- `getEps()` — metoda zwraca względną normę residuum obliczonego rozwiązania $\mathbf{x}^{sol}$, zgodnie z wzorem $\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$;
- `getResid()` — metoda zwraca normę residuum rozwiązania $\|r\| = \|\mathbf{b} - \mathbf{A}\mathbf{x}^{sol}\|$;
- `getIter()` — metoda zwraca ilość wykonanych iteracji metody;
- `getTime()` — metoda zwraca czas, jaki zabrało wywołanie metody iteracyjnej do rozwiązania z zadanym błędem danego URL.

3.7. Klasa implementujące macierze

Z punktu widzenia efektywności obliczeń MES, kluczową strukturą danych, mającą istotny wpływ na

czas obliczeń oraz na zużycie pamięci, jest klasa implementująca macierz. Projektując tę strukturę danych oraz algorytmy na niej operujące trzeba przede wszystkim uwzględnić specyfikę metod Kryłowa, jakich używamy do rozwiązywania układów równań liniowych, formułowanych dla rozwiązania procesu spęczania.

Macierz układu równań liniowych dla rozpatrywanego problemu jest macierzą macierzą rzadką i symetryczną.

Najważniejszą operacją wykonywaną na macierzy jest operacja mnożenia macierzy przez wektor. Jest to także najkosztowniejsza składowa każdej metody iteracyjnej rozwiązywania URL. Z tego też względu, projektowana struktura danych musi być zaprojektowana pod kątem optymalizacji wykonania mnożenia macierzy przez wektor.

Dla projektowanego solwera zaimplementowano dwie klasy implementujące macierze:

1. klasę `PSparSymMatrix`, która implementuje symetryczną macierz rzadką, w której nie zdefiniowano wewnętrznej struktury; klasa ta jest używana przez klasę `FEMHeatSolver` implementującą solwer dla procesu termomechanicznego;
2. klasę `PDirSymMatrix`, która jest klasą pochodną klasy `PSparSymMatrix`; klasa ta jest używana przez klasę `FEMStructSolver` implementującą solwer dla procesu termomechanicznego; zdefiniowano w niej wewnętrzną strukturę, co jest wykorzystywane do obliczeń związanych z uwarunkowaniem wstępnym dla metod Kryłowa; ponadto klasa ta umożliwia wprowadzenie do URL warunków brzegowych Dirichleta.

3.7.1. Klasa `PSparSymMatrix`

Wewnętrzną strukturę macierzy rzadkiej zaimplementowano w oparciu o format MSR (ang. *Modified Sparse Row*) (Saad, 2000). Zgodnie ze standardem języka C++ dotyczącym indeksowania tablic, indeksowanie n wierszy i kolumn macierzy o rozmiarze $n \times n$ przebiega od indeksu równego 0 do indeksu równego $n-1$. Macierz przechowywana jest w dwóch wektorach, odpowiednio typu `double` i `long`:

1. `pVal` — wektor do przechowywania wartości niezerowych elementów macierzy,
2. `pCol` — wektor do przechowywania indeksów kolumn dla pozadiagonalnych elementów macierzy oraz indeksów wskazujących, gdzie zaczynają się pozadiagonalne elementy kolejnego wiersza macierzy.

Ponieważ elementy diagonalne macierzy będą wykorzystywane częściej (ze względu na użyty *preconditioner*) będziemy przechowywać oddzielnie elementy diagonalne i pozadiagonalne. W pierwszych n elementach wektora `pVal` znajdują się kolejne wartości elementów diagonalnych macierzy. Pozycja o indeksie n w wektorze `pVal` pozostanie niewykorzystana. Począwszy od pozycji o indeksie $n+1$, w wektorze `pVal` będą przechowywane wierszami kolejne wartości pozadiagonalne. Dla wszystkich elementów pozadiagonalnych, odpowiadająca im pozycja w wektorze `pCol` będzie zawierać numer kolumny, do której należy dany element.

W programie wykorzystane są dwa szczegółowe formaty dla struktury danych opisujących macierz rzadką, które różnią się sposobem wypełnienia części pozadiagonalnej obu wektorów oraz interpretacją wartości w wektorze `pCol` na pierwszych n pozycjach:

1. Format spakowany `MAT_PACKED_HALF` — używany przez solwer iteracyjny, gdzie niezerowe elementy pozadiagonalne wypełniają nieprzerwanie tę części wektorów `pVal` i `pCol`, które zaczynają się od pozycji o indeksie $n+1$. Macierz jest przekształcana do tego formatu z formatu `MAT_UNPACKED_HALF` przez wywołanie metody `CompressMatrix()`. Przykładową macierz przechowywaną w takim formacie przedstawia rysunek 3(c).
2. Format niespakowany `MAT_UNPACKED_HALF` — używany do składania globalnej macierzy URL. Te części wektorów elementów `pVal` i `pCol`, w których przechowywane są informacje o elementach pozadiagonalnych podzielone są na n równych części (segmentów), odpowiadających poszczególnym wierszom macierzy. Szerokość segmentu jest jednym z parametrów konstruktora klasy `PSparSymMatrix` i musi być dobrana tak, aby zmieścić wszystkie niezerowe elementy pozadiagonalne dla każdego wiersza macierzy. Maksymalną szerokość pasma `nWidth` da się wyznaczyć *a priori* na podstawie topologii siatki MES oraz ilości stopni swobody przypadających na jeden węzeł siatki. Przykładową macierz przechowywaną w takim formacie przedstawia rysunek 3(b).

Zastosowanie dwóch formatów zapisu macierzy rzadkiej, mimo iż konieczna jest dodatkowa transformacja jednego formatu do drugiego, jest istotne z punktu widzenia czasu obliczeń: użycie formatu niespakowanego jest korzystne w procesie budowy globalnej macierzy układu równań, zaś formatu spakowanego — dla procedury mnożenia macierz-wektor.

Dostęp do elementu (i,j) macierzy zapewnia specjalnie zaimplementowany w klasie `PSparSymMatrix` dwuargumentowy `operator()(long, long)`. Operator ten jest używany tylko w trakcie budowy globalnej macierzy układu.

Wstawiając nowy element do macierzy zapisanej w formacie skompresowanym, musielibyśmy za każdym razem przesuwać wszystkie pozycje znajdujące się na prawo od pozycji k, co znacznie podniosłoby koszty obliczeń. W zaprezentowanym rozwiązaniu, jedyną istotną operacją jest przeszukanie segmentu, co oznacza przeglądnięcie najwyżej `nWidth` pozycji. Ewentualne wstawienie nowego elementu jest bardzo szybkie, ponieważ dysponujemy bezpośrednio indek-

$$\begin{bmatrix} 2 & 0 & -1 & 1 \\ 0 & 4 & 2 & 0 \\ -1 & 2 & 1 & 0 \\ 1 & 0 & 0 & 3 \end{bmatrix}$$

(a) Przykładowa macierz. (*Exemplary matrix.*)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
pVal	2	4	1	3	*	·	·	·	·	·	·	−1	2	·	1	·	·
pCol	5	8	13	15	·	·	·	·	·	·	·	0	1	·	0	·	·
	diag					0			1			2			3		

(b) Postać nieskompresowana (*Compressed matrix*).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
pVal	2	4	1	3	*	−1	2	1	·	·	·	·	·	·	·	·	·
pCol	5	5	5	7	8	0	1	0	·	·	·	·	·	·	·	·	·
	diag					2		3									

(c) Postać skompresowana. (*Uncompressed matrix*)

Rysunek 3: Struktura macierzy rzadkiej zaimplementowana w klasie PSparSymMatrix. (*Internal structure of sparse matrix implemented within class PSparSymMatrix.*)

sem pierwszego wolnego miejsca w segmencie i (przechowywanego w wektorze pCol na pozycji i).

Z drugiej strony, dla procedury wykonującej mnożenie macierz-wektor korzystne jest dosunięcie wszystkich niezerowych elementów pozadiagonalnych do siebie i w lewo. Jest to związane z zapewnieniem większej lokalności danych (ang. *data locality*) — w trakcie działania procedury mnożącej poszczególne wiersze macierzy przez dany wektor, kolejne wiersze następują bezpośrednio po sobie i tak też są ładowane do pamięci podręcznej *cache*. Zwiększa się zatem ilość trafień (ang. *cache hit rate*), a co za tym idzie — wydajność obliczeń. Algorytm mnożenia macierz rzadka-wektor przedstawiamy poniżej:

- `for((i = 0; i < n; ++i))` — dla elementów diagonalnych
  - `y[i] = pVal[i] * x[i];`
- `for((i = 0; i < n; ++i))` — dla elementów pozadiagonalnych
  - `y[i] += pVal[k] * x[pCol[k]];`
  - `y[pCol[k]] += pVal[k] * x[i];`

Doświadczenia numeryczne, przeprowadzone na wczesnych etapach projektowania klasy PSparSymMatrix wykazały, że dosunięcie elementów kolejnych wierszy do siebie zwiększa znacząco wydajność procedury (nawet o ok. 40%). Zadecydowało to o użyciu dwóch różnych formatów przechowywania macierzy.

W klasie PSparSymMatrix zaimplementowano także dwie metody: PrecondSSOR_U() oraz PrecondSSOR_L(), wykonujące operację uwarunkowania wstępnego dla zadanego wektora $\mathbf{v}$ metodą SSOR ($\mathbf{Mz} = \mathbf{v}$, gdzie $\mathbf{M} = (\mathbf{D} + \omega\mathbf{L})\mathbf{D}^{-1}(\mathbf{D} + \omega\mathbf{L}^T)$), zgodnie z wzorami:

1. $(\mathbf{D}+\omega\mathbf{L})\mathbf{y} = \mathbf{v}$ — procedura PrecondSSOR_L()
2. $\mathbf{D}^{-1}(\mathbf{D} + \omega\mathbf{L}^T)\mathbf{z} = (\mathbf{I} + \omega\mathbf{D}^{-1}\mathbf{L}^T)\mathbf{z} = \mathbf{y}$ — procedura PrecondSSOR_U().

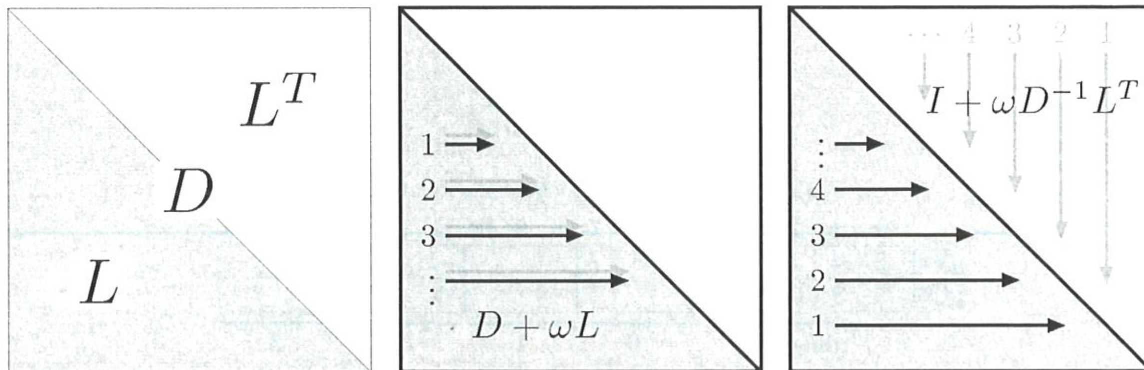
W obu przypadkach sprowadza się to do rozwiązania układu równań z macierzą trójkątną: w pierwszym — dolną, a drugim — górną.

Warto zaznaczyć, iż w przypadku uwarunkowania wstępnego metodą SSOR nie jest konieczna budowa nowej macierzy, a co za tym idzie jej osobne przechowywanie. Obie metody, PrecondSSOR_U() oraz PrecondSSOR_L(), zaimplementowano *in situ*, tzn. operują one tylko na macierzy URL nie wykorzystując żadnej dodatkowej struktury danych do przechowywania macierzy (z wyjątkiem wektora $\mathbf{y}$).

Cała operacja uwarunkowania wstępnego wektora $\mathbf{v}$ składa się z więc dwóch kroków: wywołania procedury PrecondSSOR_L() dla wektorów $\mathbf{v}$ i $\mathbf{y}$, a następnie wywołania procedury PrecondSSOR_U() dla wektorów $\mathbf{y}$ i $\mathbf{z}$. Oba algorytmy zaprojektowano tak, aby rozwiązać każdy z powyższych układów równań tylko w jednym przejściu macierzy rzadkiej. Klasa PSparSymMatrix jest używana bezpośrednio przez klasę FEMHeatSolver do przechowywania macierzy $\mathbf{A}$ oraz pomocniczej macierzy $\mathbf{Q}$. Stanowi on także klasę podstawową dla klasy PDirSymMatrix, którą omówimy poniżej.

3.7.2. *Klasa* PDirSymMatrix

Dla części solwera MES symulującej część mechaniczną procesu spęczania zaprojektowano specjalną klasę PDirSymMatrix implementującą specyficzną macierz rzadką, jaka powstaje na skutek linearyzacji rozpatrywanego problemu wariacyjnego. Klasę tę zbudowano w oparciu o klasę PSparSymMatrix, która opisuje ogólną macierz rzadką, doprecyzowując pewne — istotne z punktu widzenia obliczeniowego — własności oraz metody operujące na tej macierzy.



(a) Przechowywana jest macierz trójkątna dolna **L** oraz diagonala **D**. (*Stored diagonal and lower triangular part only.*)

(b) Metoda `PrecondSSOR_L()` — kolejność przeglądania wierszy. (*`PrecondSSOR_L()` method — ordering of processed rows*)

(c) Metoda `PrecondSSOR_U()` — kolejność przeglądania wierszy. (*`PrecondSSOR_U()` method — ordering of processed rows*)

Rysunek 4: Symetryczna macierz rzadka implementowana przez klasę `PSparSymMatrix`. Czarne strzałki pokazują kierunek fizycznego, a szare — logicznego przeglądania elementów macierzy układu. (*Symmetric sparse matrix implemented by class `PSparSymMatrix`. Arrows show physical (black) and logical (gray) ordering of processed elements.*)

Istnieją dwie główne przesłanki, które potwierdzają słuszność tej koncepcji:

1. Po pierwsze, do układu równań powstałego w procesie linearyzacji potrzeba wprowadzić warunki brzegowe Dirichleta (dotyczą one jedynie pola prędkości). Aby wykonać jak najefektywniej tę operację, która w swej istocie dokonuje zmian w samej strukturze macierzy oraz w wektorze prawej strony URL, konieczne jest, aby metoda, która ją wykonuje, pracowała bezpośrednio i niskopoziomowo na poszczególnych składowych macierzy (a więc tablicach `pVal` i `pCol`). Metoda taka, musi być zatem składową klasy, przez co będzie miała dostęp do prywatnych atrybutów klasy.

2. Po drugie, rozpatrywana macierz posiada strukturę blokową (patrz równanie 7 oraz rysunek 5(a)) o określonych własnościach matematycznych, co — jak wykazała praca Blanka i in. (1999) — daje asumpt do zastosowania pewnych klas preconditionerów, które w połączeniu z metodami Kryłowa dają efektywne solwery rozwiązujące układy równań liniowych tej klasy. Preconditionery takie powielają wyjściową strukturę macierzy układu równań, tak więc wydaje się uzasadnione, aby metoda implementująca preconditioner tego typu była zaimplementowana w klasie opisującej blokową macierz rzadką.

Reasumując, w klasie `PDirSymMatrix` została zdefiniowana metoda `ImposeStruct Dirichlet()` wprowadzająca warunki brzegowe Dirichleta oraz dwie metody: `Precond SSOR_Li()` i `PrecondSSOR_Ui()` wykonujące operację uwarunkowania wstępnego dla zadanego wektora.

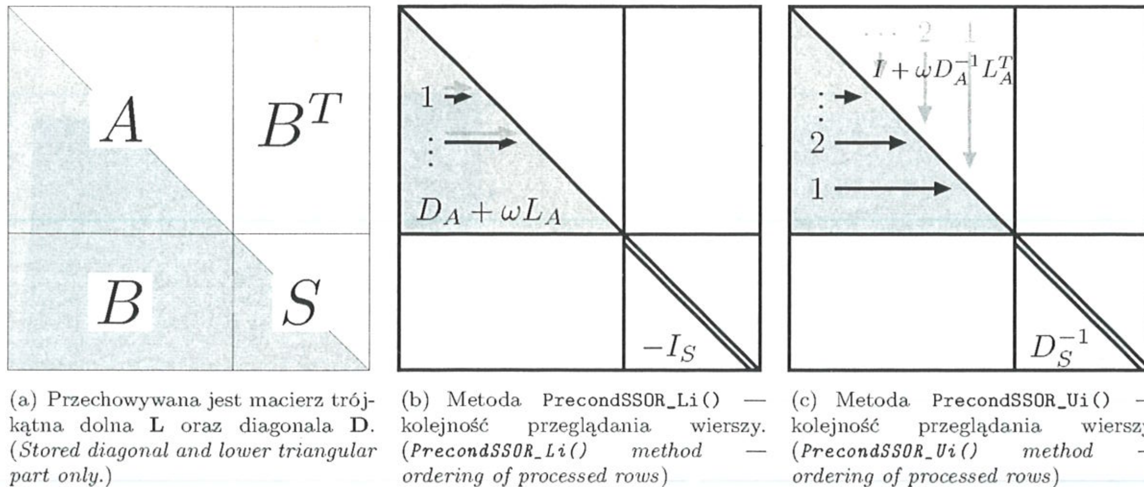
Z warunków postawionego zadania wynika, iż nałożenie warunku brzegowego Dirichleta na odpowiedni stopień swobody w układzie równań powstałym w procesie linearyzacji sprowadza się na wymuszeniu zerowej zmiany prędkości (bo w rozpatrywanym zadaniu mamy do czynienia tylko z kinematycznymi warunkami brzegowymi). Tak więc, jeżeli na i-ty stopień swobody nałożony jest warunek Dirichleta, wtedy zerujemy cały i-ty wiersz oraz całą i-tą kolumnę macierzy, elementowi diagonalnemu nadajemy wartość 1, a na i-tą pozycję w wektorze prawej strony oraz na i-tą pozycję w wektorze niewiadomych wstawiamy wartość 0.

Procedura wprowadzająca warunki brzegowe do układu działa w ten sposób, iż przegląda po kolei wiersze macierzy sprawdzając, czy wiersz ten ma wyzerować (jeśli nałożono warunek brzegowy), a jeśli nie, to zeruje w tym wierszu wszystkie te elementy, które należą do kolumn odpowiadających stopniom swobody, na które nałożono warunki brzegowe.

Procedury `PrecondSSOR_Li()` oraz `Precond SSOR_Ui()` wykonujące wspólnie operację uwarunkowania wstępnego dla zlinearyzowanego URL działają następująco:

- dla podmacierzy **A** — uwarunkowanie wstępne metodą SSOR (wykonują operacje analogiczne jak `PrecondSSOR_L()` i `PrecondSSOR_U()`);
- dla podmacierzy **S** — uwarunkowanie wstępne metodą Jacobiego.

Ponieważ, z założenia, przechowujemy tylko dolną trojkątną część macierzy układu, a mamy ograniczyć operację preconditioningu metodą SSOR tylko do podmacierzy **A**, wystarczy zauważyć, że wszystkie przechowywane wiersze całej macierzy układu, poczynając od wiesza o indeksie 0, aż do wiersza odpowiadającego ostatniemu stopniowi swobody opisującemu pole prędkości, są jednocześnie wierszami części trójkątnej dolnej podmacierzy **A** (porównaj rysunki 5a i 5b). Dzięki temu operacja preconditiongu może być wykonana bezpośrednio na tej części macierzy układu: algorytmy `PrecondSSOR_L()` i



(a) Przechowywana jest macierz trójkątna dolna **L** oraz diagonala **D**. (*Stored diagonal and lower triangular part only.*)

(b) Metoda `PrecondSSOR_Li()` — kolejność przeglądania wierszy. (*`PrecondSSOR_Li()` method — ordering of processed rows*)

(c) Metoda `PrecondSSOR_Ui()` — kolejność przeglądania wierszy. (*`PrecondSSOR_Ui()` method — ordering of processed rows*)

Rysunek 5: Symetryczna macierz rzadka implementowana przez klasę `PDirSymMatrix`. Czarne strzałki pokazują kierunek fizycznego, a szare — logicznego przeglądania elementów macierzy układu. (*Symmetric sparse matrix implemented by class `PDirSymMatrix`. Arrows show physical (black) and logical (gray) ordering of processed elements.*)

`PrecondSSOR_U()` są wykonywane tylko dla wierszy podmacierzy **A**. Sytuacja taka nie byłaby możliwa, gdybyśmy przechowywali nie część trójkątną dolną ale górną — wtedy w pierwszych wierszach całej macierzy układu, poczynając od wiesza o indeksie 0, aż do wiersza odpowiadającego ostatniemu stopniowi swobody opisującemu pole prędkości, przechowywalibyśmy nie podmacierz **A**, ale podmacierz $\mathbf{A} + \mathbf{B}^T$.

3.8. Implementacja Solwera MES

Wszystkie składowe funkcjonalne solwera MES, a zatem opisane w poprzednich rozdziałach struktury danych oraz algorytmy są połączone w głównej funkcji programu `Compress3D`. Następujące podstawowe składowe solwera, instancje klas: `Grid`, `TestingMachine`, `MaterialModel`, `FEMHeatSolver` oraz `FEMStructSolver` zdefiniowano jako obiekty globalne, dostępne poprzez wskaźniki i tworzone dynamicznie biorąc pod uwagę parametry odczytywane z plików konfiguracyjnych.

Jedyną funkcją pomocniczą, zdefiniowaną w głównej części programu jest funkcja `ReadDataFile()`, która wczytuje plik z danymi opisującymi nazwę pliku z siatką, dane materiałowe oraz dane procesu, a następnie — po dokonaniu jego parsingu — powołuje do życia obiekty klas: `Grid`, `TestingMachine` oraz `MaterialModel`, przygotowując struktury danych dla obu solwerów MES.

Po wczytaniu siatki, tworzone są instancje klas implementujących solwery MES, a następnie obliczany jest rozkład prędkości początkowych. Pętla po krokach czasowych obejmuje: procedurę linearyzacji, w wyniku której otrzymujemy nowe pole prędkości i pole ciśnień, obliczenie siły i odkształcenia próbki, obliczenie pomocniczego rozkładu temperatury, modyfikację położenia węzłów siatki, obliczenie ostatecznego rozkładu temperatury oraz przeliczenie pola prędkości. Pętla ta kończy działanie po osiągnięciu przez próbkę żądanego odkształcenia.

4. PRZYKŁADOWA SYMULACJA

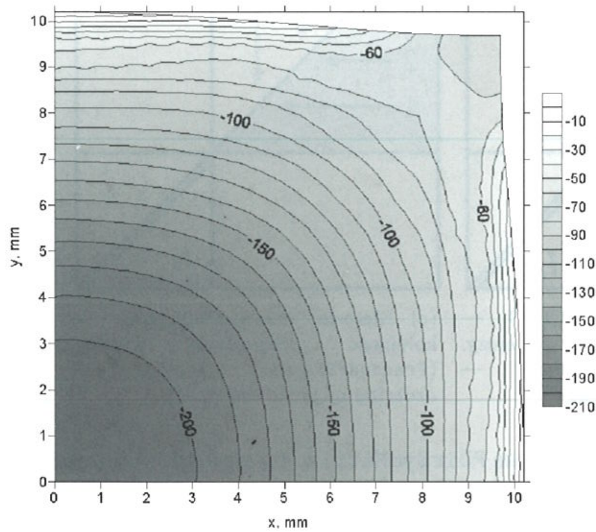
Przeprowadzono symulację spęczania próbki sześciennej o wymiarach $15 \times 15 \times 15$mm. Przyjęto sztywno-plastyczny model konstytutywny (współczynnik wrażliwości w równaniu Nortona-Hoffa $m = 0$), krzywą umocnienia $\sigma_p = 20 + 370\varepsilon_i^{0.38}$MPa, współczynnik tarcia $m = 0.2944$. Proces przeprowadzono dla stałej prędkości spęczania równej 1mm/s. Próbkę spęczono o 40%. Rezultaty symulacji przedstawiono na rysunkach 7, 6, 8 i 9.

5. WNIOSKI

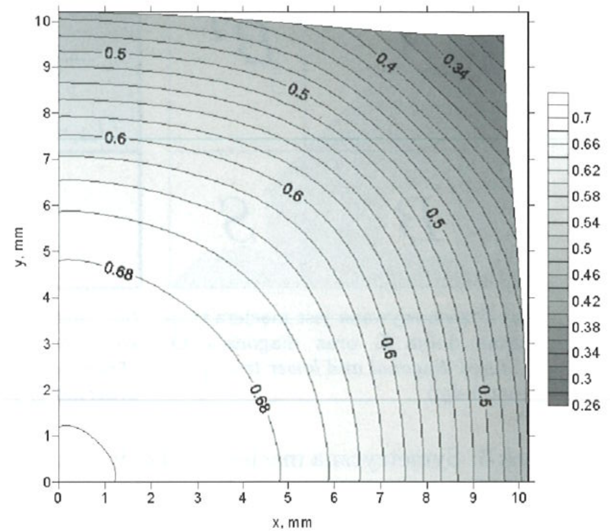
W niniejszej pracy przedstawiono model fizyczny i obliczeniowy dla procesu spęczania próbki oraz jeden z wielu możliwych sposobów opisu i implementacji tych modeli z użyciem metodologii obiektowej i języka C++.

Model fizyczny oparto o quasi-stacjonarny model przepływu nieściśliwego płynu o silnie nieliniowej lepkości, zdefiniowanej prawem Nortona-Hoffa. W celu użycia różnorodnych typów elementów skończonych do tak zdefiniowanego sformułowania mieszanego zastosowano stabilizację SUPG. Zbieżność linearyzacji modelu wsparto technikami: subinkrementacji, gdzie minimalizujemy normę residualną, oraz progresywnej strategii linearyzacji modelu konstytutywnego. Dzięki temu uzyskano zbieżność metody dla wszystkich testowanych geometrii próbek, własności materiału oraz warunków fizycznych testów.

Zastosowanie podejścia obiektowego do analizy, projektowania i implementacji obiektowej rozpatry-

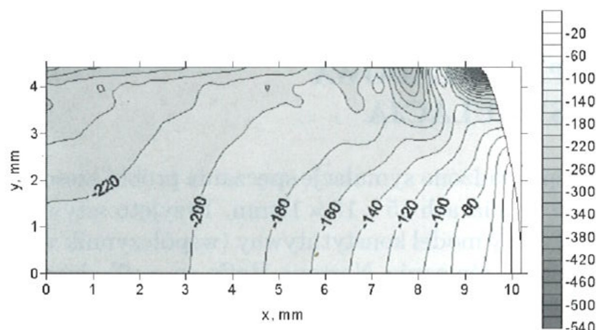


(a) Naprężenie średnie p, MPa. (*Mean stress*)

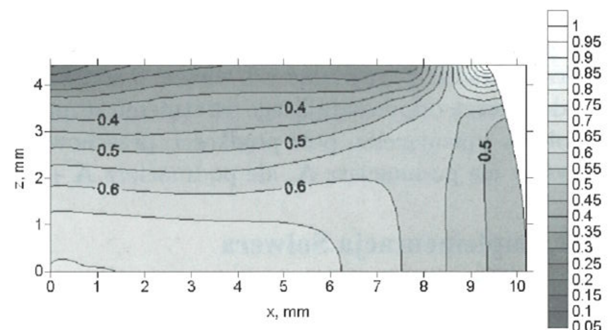


(b) Intensywność odkształcenia ε_i. (*Strain intensity*)

Rysunek 6: Rozkłady naprężeń i odkształceń dla płaszczyzny przekroju OXY. (*Stress and strain distribution for OXY section plane*)

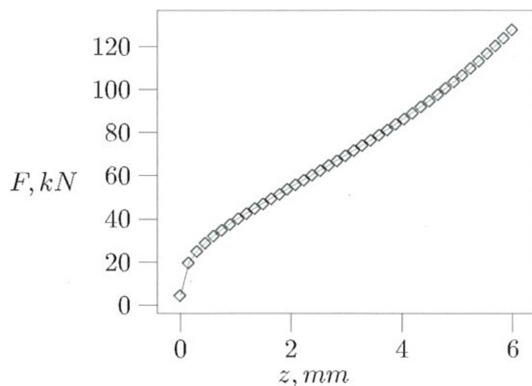


(a) Naprężenie średnie p, MPa. (*Mean stress*)

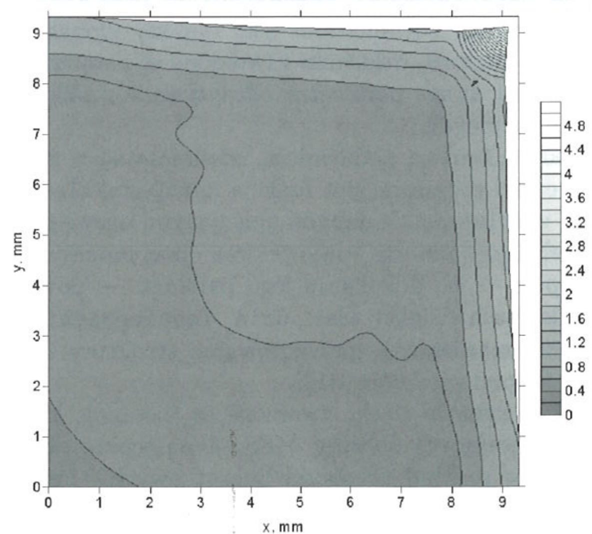


(b) Intensywność odkształcenia ε_i. (*Strain intensity*)

Rysunek 7: Rozkłady naprężeń i odkształceń dla płaszczyzny przekroju OXZ. (*Stress and strain distribution for OXZ section plane*)



Rysunek 8: Siła w funkcji przemieszczenia. (*Load versus displacement*)



Rysunek 9: Intensywność odkształcenia ε_i na powierzchni styku narzędzia z materiałem. (*Strain intensity for contact surface*)

wanego modelu obliczeniowego dało w rezultacie zmodularyzowany, przejrzysty, elastyczny i efektywny kod solwera MES. Zaprojektowane algorytmy i struktury danych ujęte w poszczególne klasy oraz hierarchie klas mogą być być dalej rozwijane, bez większych ingerencji w kod pozostałych klas. Dzięki zastosowaniu takiego podejścia, skrócił się proces implementacji programu, dzięki szybszemu i bardziej precyzyjnemu wychwytywaniu błędów. Ponadto, poszczególne składowe (klasy) zaprojektowano tak, aby mogły być używane przy implementacji innych programów tego typu, co wydatnie skróci czas ich powstania.

Praca wykonana w ramach projektu KBN nr 4T08B 009 24.

LITERATURA

Blank, H., Rudgyard, M., Wathen, A.J., 1999, Stabilised finite element methods for steady incompressible flow, Comp. Meth. Appl. Mech. Eng., 174:91-105.

Brezzi, F., Fortin, M., 1991, Mixed and Hybrid Finite Element Methods, Springer-Verlag.

Chen, C.C., Kobayashi, S., 1978, Rigid-plastic-finite-element analysis of ring compression. Application of Numerical methods to Forming Processes, ASME, ADM, 28:163-174.

Dubois-Pelerin, Y., Zimmermann, T., Bomme, P., 1992, Object-oriented finite element programming: II, A prototype programm in Smalltalk, Comp. Meth. Appl. Mech. Eng., 98.

Dubois-Pelerin, Y.,Zimmermann, T., 1993, Object-oriented finite element programming: III, An efficient implementation in C++, Comp. Meth. Appl. Mech. Eng., 108.

Elman, H.C., Silvester, D.J., Wathen, A.J., 1996, Iterative Methods for Problems in Computational Fluid Dynamics, Technical Report CS-TR-3675.

Hartley, P. and Pillinger, I. and Sturgess, C. ed., 1992., Numerical Modelling of Material Deformation Processes, Springer-Verlag.

Hughes, T.J.R., Franca, L.P., 1987, A new finite element method for computational fluid dynamics: VII. The Stokes problem with various well-posed boundary conditions: Symmetric formulations that converge for all velocity/pressure spaces, Comp. Meth. Appl. Mech. Eng., 65:85-96.

Hughes, T.J.R., Franca, L.P., Balestra, M., 1986, A new finite element formulation for computational fluid dynamics: V. Circumventing the Babuska-Brezzi condition: A stable Petrov-Galerkin formulation of the Stokes problem accomodating equal-order intrpolations, Comp. Meth. Appl. Mech. Eng., 59:85-99.

Jansen, K., Collis, S., Whiting, C., Shakib, F., 1999, A better consistency for low-order stabilized finite element methods.

Maniaty, A.M., Liu, L., Klaas, O., Shephard, M.S., 2001, Stabilized finite element method for viscoplastic flow: formulation and a simple progressive solution strategy, Comp. Meth. Appl. Mech. Eng., 190:4609-4625.

Saad, Y., 2000, Iterative Methods for Sparse Linear Systems, SIAM.

Wagoner, R.H., Chenot, J.L., 2001, Metal Forming Analysis, Cambridge University Press.

Zimmermann, T., Dubois-Pelerin, Y., Bomme, P., 1992, Object-oriented finite element programming: I, Governing principles, Comp. Meth. Appl. Mech. Eng., 98.

Artykuł otrzymano 30 grudnia 2004 r.