TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: B 2612 – Elektrotechnika a informatika

Studijní obor: 2612R011 – Elektronické informační a řídicí systémy

Efektivní řešení problémů topologie diskretizačních sítí

Efficient problem solving of topology diskretization meshes

BAKALÁŘSKÁ PRÁCE

Autor: **Martin Kopeček**

Vedoucí bakalářské práce: Ing. Dalibor Frydrych, Ph.D.

Konzultant: Ing. Milan Hokr, Ph.D.

V Liberci 17. 5. 2007

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Ústav nových technologií
a aplikované informatiky Akademický rok: 2006/2007

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jméno a příjmení: Martin Kopeček
studijní program: B 2612 – Elektrotechnika a informatika

obor: 2612R011 – Elektronické informační a řídicí systémy

Vedoucí katedry Vám ve smyslu zákona o vysokých školách č.111/1998 Sb. určuje tuto bakalářskou práci:

Název tématu:

Efektivní řešení problémů topologie diskretizačních sítí

Zásady pro vypracování:

1. Seznamte se základy metody konečných prvků se zřetelem na strukturu výsledné soustavy lineárních rovnic
2. Seznamte se základy objektového návrhu numerických modelů
3. Vytvořte třídy řešící úlohy topologie, navrhněte základní datové struktury pro uložení a metody, pracující nad těmito datovými strukturami
4. Funkčnost tříd ověřte na jednoduchých úlohách

Rozsah grafických prací: dle potřeby dokumentace

Rozsah průvodní zprávy: cca 40 stran

Seznam odborné literatury:

[1] B.Eckel: Myslíme v jazyku Java, Grada Publishing, Praha, 2001, ISBN: 80-247-9010-

[2] O.C.Zienkiewicz, R.L.Taylor: Finite Element Method (5th Eition) Volume 1 – The basis, Elsevier, 2000, ISBN 0-7506-5049-4

[3] K.Rektorys: Variační metody, Academia Praha 1999, ISBN 80-200-0714-8

[4] E.F. Kaasschieter, Preconditioned Conjugate Gradients and Mixed-Hybrid Finite Elements for the Solution of Potential Flow Problems, Ph.D. Thesis, Delf University of Technology 1990

Vedoucí bakalářské práce: Ing. Dalibor Frydrych, Ph.D.

Konzultant: Ing. Milan Hokr, Ph.D.

Zadání bakalářské práce: **20.10.2006**

Termín odevzdání bakalářské práce: **18. 5. 2007**

....

Vedoucí katedry Děkan

V Liberci dne 20.10.2006

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé BP a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum: 17. 5. 2007

Podpis:

Poděkování

Děkuji Ing. Daliborovi Frydrychovi, Ph.D. za hodnotné rady, cenné připomínky a odborné vedení během mé práce.

Anotace

Martin Kopeček, Bakalářská práce

Liberec, TUL, Fakulta Mechatroniky a mezioborových inženýrských studií,

Ústav nových technologií a aplikované informatiky

Stran: 41 (+4)
Obrázků: 7 (+1)
Tabulek: 10

V literárním rozboru je v první části kladen důraz především na přehled základních metod využívaných pro modelování a simulaci v počítačových programech. Podrobně je rozebrána Metoda konečných prvků. Druhá část se zabývá problémy spojenými s návrhem prvků sítě a jejich realizací pomocí programu Gmsh. Ve třetí, experimentální části, jsou ukázány teoretické předpoklady a vytvořeny funkční metody v programovacím jazyce Java, které nalézají vztahy mezi různými entitami, jakými jsou uzly, elementy a hrany. V závěru každé problematiky je ukázán výsledný funkční výstup na konzoli.

Klíčová slova:

MKP; Java; Gmsh; prvek; uzel; hrana

Annotation

Martin Kopeček, Baccalaureate work

Liberec, TUL, Faculty of Mechatronics and Interdisciplinary Engineering Studies,

Institution of new technology and applied information science.

Pages: 41 (+4)
Pictures: 7 (+1)
Tables: 10

In a first part of literary analysis is first of all laid stress on basic methods used for modelling and simulation in computer programs. There is in detail parsed Finite Element Method. Second part is dealt with the problems involved in proposal elements meshes and theirs realization by the help of programme Gmsh. In third, experimental, part are shown theoretical possibilities and created function methods in programming language Java, which describe relations between different entities which are nodes, elements and lines. At the end of every problem is shown final working output on console.

Key words:

FEM; Java; Gmsh; element; node; line

Obsah

Seznam použitých symbolů, zkratek a pojmů

ASCII	*American Standard Code for Information Interchange*	-Americký standardní kód pro výměnu informací,osmibitová abeceda tvořící základ pro většinu dnes používaných abeced pro kódování písmen a číslic.
HTML	*HyperText Markup Language*	-Jazyk sloužící k popisu webových stránek.
ISO	*International Organization for Standardization*	-Mezinárodní normalizační organizace
Node	*Uzel*	-(nebo uzlový bod)určuje místa spojení jednotlivých prvků
Element	*Prvek*	-Anglický výraz z něhož vychází i FEM
Line	*Hrana*	-Určuje vzdálenost jednotlivých sousedních uzlů. Je součástí prvku.
Java		-objektově orientovaný programovací jazyk
Entita		-prvek, uzel, hrana
JRE	*Java Runtime Environment*	-Je prostředí pro běh programů v Javě, které stručně řečeno zahrnuje: interpret Javy a standardní knihovny.
JDK	*Java Development Kit*	-Obsahuje JRE plus překladač a další vývojové nástroje.
IDE	*Integrated development environment*	-Anglická zkratka pro vývojové prostředí.
Gmsh	*G-mesh*	-Je to automatický generátor jednorozměrných, dvourozměrných a trojrozměrných sítí konečných prvků.
Garbage collector		-Jedná se o „čistič paměti“. Odstraňuje nepotřebný objekt z paměti.
CAD	*Computer Aided Design*	-Počítačové modelování
SW	*Software*	-Programové vybavení počítače

Úvod

V dnešní době se velmi rozšířil vědní obor zabývající se počítačovým modelováním přírodních procesů. Již od 60. let se vědci věnují metodám, které napomohly k vytvoření současné podoby simulačních programů. Ty jsou dnes považovány jako silný nástroj k předpovídání a vyhodnocování výsledků u mnoha inženýrských projektů .

Tato práce řeší především problémy, které jsou spjaty s tvorbou a úpravou topologie diskretizačních sítí. Nezabývá se výslovně diskretizací dané oblasti, využívá generátor sítí Gmsh a hledá souvislosti mezi prvky takto vytvořené sítě. Okrajově se zabývá metodou konečných prvků a popisem počítačových modelů, obzvláště je ale zaměřena na tvorbu entit v síti a na jejich závislostech mezi sebou pomocí programovacího jazyka Java.

Cílem této práce je vytvořit aktivní, univerzální a obecný systém, který nám umožní vytvářet, rozšiřovat a vkládat nové prvky do modelu a automaticky vyhledávat příslušné vztahy mezi nimi. Srozumitelným způsobem budou vysvětleny použité metody v jazyce Java, poukáže se na jejich funkci a porovnají se s teoretickými hypotézami.

Předpokládá se základní znalost jazyka Java, a proto zde bude uvedena jeho hlavní syntaxe.

1 Podstata metody konečných prvků (MKP)

Základem vědeckotechnických výpočtů pro potřebu technického rozvoje jsou řešení známých diferenciálních rovnic matematické fyziky. Výpočtové postupy před nástupem počítačů používaly často velmi zjednodušujících předpokladů, výsledky analytického řešení proveditelných jen pro geometricky jednoduché oblasti a zvláštní případy okrajových podmínek, příhodných pro výpočet. Rozvoj výpočetní techniky umožnil zavedení novodobých výpočetních metod, způsobilých k řešení úloh v prakticky libovolně utvářených geometrických oblastech a při okrajových podmínkách běžně se vyskytující v technické praxi. Největší význam mezi přibližnými numerickými metodami získala metoda konečných prvků, která se brzy prosadila jak při řešení úloh z oblasti pružnosti a pevnosti, tak i z oblasti teplotních polí a obecně potenciálních úloh stacionárních i nestacionárních [3].

Hlavní myšlenkou metody je, že se nejprve diskretizuje vyšetřované těleso, tj. rozdělí se na konečný počet jednotlivých oblastí, což jsou pro rovinnou úlohu většinou trojúhelníky či čtyřúhelníky a pro prostorové úlohy čtyřstěny, pětistěny, kvádry a podobně. Pojem diskrétní je opakem pojmu kontinuální. Vhodnou volbou bázových funkcí lze tuto úlohu převést na řešení soustavy lineárních rovnic, jejíž matice je řídká, tj. obsahuje většinou nulové prvky. Řídkost matice snižuje nároky na paměť počítače a počet prováděných aritmetických operací. To nám již v současnosti umožňuje řešit obrovské soustavy až o miliónech rovnic a milionech neznámých na počítačích s paralelní architekturou. Ovšem velkým problémem je zde pořád doba výpočtu takto náročných operací [14].

Přesná matematická definice MKP (podle [3])**:**

MKP je zobecněná Ritz-Galerkinova variační metoda, užívající bázových funkcí s malým kompaktním nosičem, úzce spjatým se zvoleným rozdělením řešené oblasti na konečné prvky**.**

1.1. Metoda konečných diferencí – MKD (metoda sítí)

Metoda konečných diferencí (MKD) je vhodná pro mnohé aplikace, kde je využito její vhodnosti při aproximaci řešení spojených s teplotním přenosem. Pro použití této metody nepochybně mluví dva důležité argumenty. Prvním z nich je jednoduchost při programování a numerické realizaci a druhým důvodem je relativní jednoduchost v nelineárních matematických modelech. Mezi vady této metody však patří problém s aproximací okrajových podmínek na jednotlivých částech hranic, které nejsou vhodně použitelné na rozdílně husté sítě, zhoršení přesnosti aproximovaného řešení pro síť s různým odstupem uzlů a konečně, nezbytnost relativně hustého časového kroku. Každá geometrie musí být pro potřeby výpočtů rozdělena na síť. Prostorová (geometrická) síť je tvořena skupinou samostatných bodů v určité oblasti.

Pomocí této diferenční metody se úloha převede dle diferenciálního operátoru (nejčastěji pomocí Taylorova rozvoje) na diferenciální rovnice, podle níž se různá tělesa mohou řešit za určitých omezení - okrajové podmínky pro řešení diferenciálních rovnic [14].

1.2. Metoda hraničních prvků - MHP

Metoda hraničních prvků je novější metodou pro řešení okrajových počátečních problémů. Její popis by překračoval rámec této práce [3].

1.3. Metoda konečných prvků - MKP

Metoda konečných prvků představuje moderní, vysoce efektivní numerickou metodu pro řešení technických a vědeckých úloh jak již bylo popsáno výše. V současnosti je považována za jednu z nejúčinnějších přibližných metod pro řešení problémů popsaných **diferenciálními rovnicemi** a patří mezi vůbec nejrozšířenější metody v tomto odvětví. Na jejích základech je postaveno bezmála **95%** všech výpočetních programů zabývající se modelováním.

Podle literatury [10] metodu konečných prvků navrhl v roce 1943 Richard Courant, americký matematik německého původu. Zhruba o deset let později byla znovu objevena americkými inženýry při provádění pevnostních výpočtů leteckých

konstrukcí. Systematické teoretické studium MKP začalo až v šedesátých letech. V roce 1968 dokázal jako první konvergenci MKP brněnský profesor Miloš Zlámal (1924-1997). Rozvoj MKP vedl přirozeně k souběžnému vzniku více programů, postavených na bázi algoritmu MKP a vyvíjených zpočátku v univerzitním prostředí v souvislosti s řešením výzkumných úkolů. S rozšiřujícími znalostmi se stále častěji používalo vyvinutého softwaru k řešení inženýrských problémů, vycházejících přímo z požadavků průmyslové praxe. Zájem o nový výpočtový prostředek pak přirozeně vedl k rozvoji některých programů na čistě komerční bázi. S nástupem CAD –(*počítačové modelování, počítačem podporované konstruování, zkratka označující software (nebo obor) pro projektování či konstruování na počítači*) ještě zvýšil význam této metody a tvoří jeden ze základních bloků moderního počítačového navrhování. Zvlášť se vznikem 3D modelářů dochází k přímo skokové změně. Současné období lze charakterizovat jako vzájemné sbližování SW (*software*) a rostoucí zájem technických pracovníků s ohledem na komfort nabízených prostředí SW.

Značná pozornost je v současnosti věnována rozvoji nového směru v metodě konečných prvků, pro který se vžil anglický termín "domain decomposition method". Při tomto postupu je těleso rozděleno na několik oblasti, které mají relativně jednoduchý geometrický tvar, využije se toho, že na oblastech takto jednoduchého tvaru lze počítat tzv. rychlé algoritmy, čímž se značně sníží počet výpočtových operací [13].

Způsob vytváření modelu:

Toto prostředí vytváří koncový uživatel:

· Interpretace vstupních dat

· Čtení vstupních dat

· Zpracování a generování sítě

Tuto problematiku běžný uživatel dnešních moderních programů, generující sítě, neřeší:

· Prostor pro ukládání dat

· Sestavení soustavy lineárních rovnic

· Vyřešení soustavy lineárních rovnic

· Uložení výstupních dat

· Aposteriorní odhady chyby - Založené na smyslové zkušenosti, dnes poměrně moderní záležitost.

Toto prostředí využívá koncový uživatel:

· Grafické znázornění výsledků

Těchto devět bodů je implementováno v nepřeberném množství programů, které vytvářejí uživatelsky příjemné prostředí pro zadání úlohy spolu s kontrolou vstupních údajů.

Zjemňování sítě probíhá buď **interaktivně** (offline), kdy si uživatel sám volí oblasti, kde chce získat lepší aproximaci řešení, poté nechá SW vypočítat danou oblast znovu a takto tento cyklus opakuje, dokud není s výsledkem spokojen. **Adaptivní** (online) probíhá bez zásahu člověka. V tomto druhém případě počítač sám vyhodnocuje velikost chyby na jednotlivých prvcích, které pak případně dále rozděluje. Příslušná výstupní data jsou pak ve formě izolinií, různě obarvených, stínovaných či vyšrafovaných ploch.

1.3.1. Doplňující dělení MKP

V současnosti je MKP široce a podrobně rozpracovávaný vědní obor a v některé literatuře se můžeme setkat s využitím této metody ve spojitosti s následujícími obory:

a) teoretická

- formulace variačních principů, odvozování vztahů pro různé typy prvků atd.

b) matematická

- problematika vhodných numerických metod, výběr algoritmů, důkazy existence a konvergence řešení, odhad chyby řešení atd.

c) počítačová

- předzpracování - generování vstupních dat, grafické zobrazení členění, vstupní data, okrajové podmínky, zatížení, opravy a úpravy dat atd.
- zpracování - výpočet matic prvků, sestavení matic celého systému, sestavení maticových rovnic a jejich řešení atd.
- následné zpracování - výpočty závislých parametrů, výstupní soubory, grafické znázornění výsledků, výstupy výsledků na periferie atd.

d) inženýrsko problémová

- využití možnosti MKP pro konkrétní inženýrské úlohy tj. dělení tělesa na prvky, výběr typu prvku, výběr vhodného prvku pro danou úlohu, zadání potřebných vstupních údajů, volba forem výstupů atd.

Dnes jsou vyvinuty stovky typů konečných prvků a na světě existují desítky celosvětově známých programových systémů (ADINA, ANSYS, GMSH, APPLE-SAP, ASAS, ASKA, BEASY, COSMOS, CASTEM, DIAL, MARC, MSC/NASTRAN, PAFEC, SAP7, SYSTUS, SYSNOISE, TITUS, TPS10...).

2 Java2 SE

Java je objektově orientovaný programovací jazyk. Vyvinula jej firma Sun Microsystems a představila 23. května 1995. Je jedním z nejpoužívanějších programovacích jazyků na světě. Díky své přenositelnosti je používán pro programy, které mají pracovat na různých systémech. Jeho syntaxe je zjednodušenou (a drobně upravenou) verzí syntaxe jazyka C a C++. S výjimkou osmi primitivních datových typů jsou všechny ostatní datové typy objektové. Je navržen pro podporu aplikací v síti (podporuje různé úrovně síťového spojení, práce se vzdálenými soubory, umožňuje vytvářet distribuované klientské aplikace a servery). Velkou výhodou Javy je také její *hardwarová nezávislost*, neboť je překládaná do speciálního mezikódu (*bytecode*), který je na konkrétním počítači nebo zařízení (PC, handheld, mobilní telefon apod.) interpretován, příp. za běhu překládán do nativního kódu (tzv. *JIT - Just-In-Time* compilerem). Programátor tedy může napsat Javovský program například na PC pod Windows a spustit jej na PC s Linuxem, na MacIntoshi, SGI, DEC - zkrátka všude, kde je k dispozici *Java runtime*. Přestože se jedná o jazyk interpretovaný, není ztráta výkonu významná, neboť překladače pracují v režimu „právě včas“ a do strojového kódu se překládá jen ten kód, který je opravdu zapotřebí. Také podporuje zpracování vícevláknových aplikací. Velice pěkně se v něm pracuje, je snadno čitelný (např. i pro publikaci algoritmů) a přímo vyžaduje ošetření výjimek a typovou kontrolu. Prostředí pro práci s metodou konečných prvků bylo zvoleno JDK 1.5 (JDK 5.0) STANDARD EDITION(SE)- verze září 2004 [4],[9].

Pro psaní programů je k dispozici celá řada vývojových prostředí (IDE), a to jak freewareových, tak i komerčních. V této práci bylo použito vývojové prostředí NetBeans 5.0 viz níže.

3 NetBeans 5.0

NetBeans je otevřený a dostupný kód, a také legální licence softwaret s rozsáhlou uživatelskou základnou, komunitou vývojářů a s více jak 100 partnery po celém světě. Pod firmu Sun Microsystems, která je hlavním sponzorem projektu, přešel na základě akvizice stejnojmenné české společnosti v říjnu 1999. Vývojové prostředí NetBeans IDE je nástroj, pomocí kterého programátoři mohou psát, překládat, ladit a šířit programy. Vývojové prostředí je vytvářeno v jazyce Java. Kromě toho také existuje velké množství modulů, které toto vývojové prostředí rozšiřují. Vývojové prostředí NetBeans je bezplatně šířený produkt, který je možné používat bez jakýchkoliv omezení[13].

4 Gmsh

Je to automatický generátor jednorozměrných, dvourozměrných a trojrozměrných sítí konečných prvků s vytvářením možností předzpracování a následného zpracování. Jeho cíl je poskytovat jednoduchý nástroj pro akademické problémy pokročilými vizualizačními schopnostmi.

Gmsh je postavený ze čtyř jednotek: geometrie, síť, řešení a následné zpracování. Specifikace každého vstupu k těmto jednotkám je udělaná buď interaktivně-používání grafického uživatelského rozhraní, nebo v ASCII používání textových souborů - Gmsh má vlastní skriptovací jazyk [11].

4.1. Geometrické modely a MKP v Gmsh

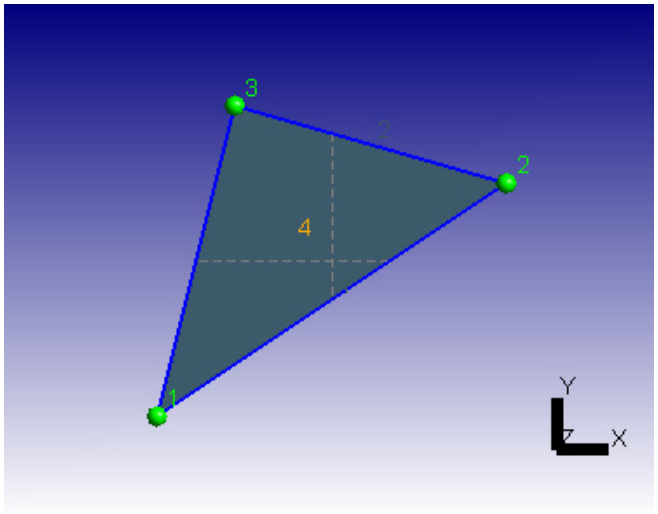
Konečně-prvkový model jsem v této práci rozdělil dvou typů entit: z elementů a uzlů popřípadě i třetí entity stran elementů[1]. Elementy ve svém souhrnu představují oblast v prostoru, která reprezentuje modelované těleso. V oblasti zkoumaní daného problému si program přizpůsobuje hustotu elementů podle složitosti daného modelu. Čím více elementový model se vytvoří a čím více je na něm oblastí s jemnější strukturou, tím je program náročnější na výpočet. V dnešní době se dá již pracovat s mnoha základními tvary elementů. V této práci se zkoumají dva považované za nejzákladnější. Jsou jimi trojúhelník a čtyřúhelník. Výhodnost těchto prvků spočívá u každého v něčem jiném. Jako dobrý příklad si můžeme zvolit kouli (fotbalový míč). Jestliže pro diskretizaci zvolíme méně hustou síť, nemusíme se bát takový model pokrýt čtyřúhelníky. Ovšem tento „míč“ bude kostrbatý. Je-li zvolena struktura z trojúhelníků, můžeme dosáhnout opravdu jemného vymodelování a „vysíťování“ tak, že se daleko více přibližujeme k popsání skutečného tělesa. Vliv na to má především rozměr právě trojúhelníkového elementu, který má poloviční rozměr než-li čtyřúhelník.

Mezi nevhodné geometrie lze považovat u trojúhelníkového elementu případ, kdy je jeden nebo dva z úhlů blízký nule, u čtyřúhelníkového třeba velmi protáhlý nebo hodně zkosený lichoběžník. Elementy nevhodného tvaru mohou typicky vzniknout při zjemňování sítí ve specifické části modelu, např. v okolí složitějšího geometrického útvaru.

[1] Stranami, neboli hranami elementů se budeme zabývat později.

4.1.1. Tvorba jednoduchých prvků modelu

Základní způsob popisu a zadání MKP modelu vychází z toho, že geometrie elementu je plně určena polohou uzlových bodů. Stačí tedy zadat souřadnice uzlů a pro každý element vypsat uspořádaný seznam jeho uzlů, jako je tomu v obrázku.

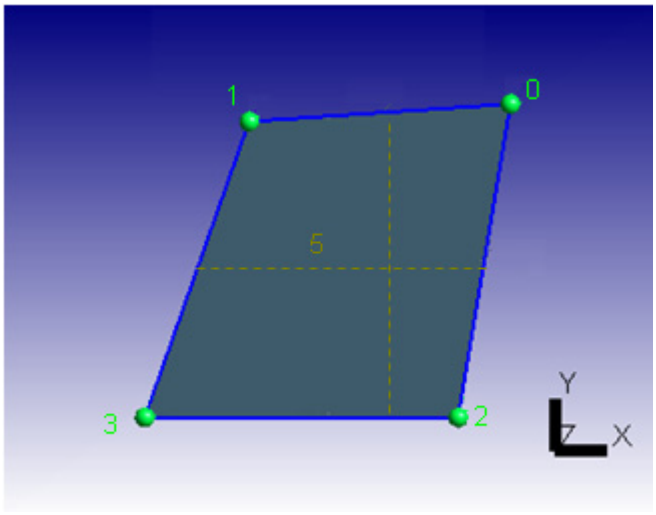


Tento jeden element tvořený 3-mi uzly by tedy mohl být popsán:

Uzel 1 (0 , 0)
Uzel 2 (3 , 2)
Uzel 3 (0.5 , 2.5)

Element: 4 (1,2,3)

Obr.1 *Trojúhelníkový element*



Tento druhý element tvořený 4-mi uzly by tedy mohl být popsán:

Uzel 0 (3 , 4.2)
Uzel 1 (1 , 4)
Uzel 2 (3 , 0)
Uzel 3 (0 , 0)

Element: 5 (0,1,2,3)

Obr.2 *Čtyřúhelníkový element*

Je zřejmé, že nutnou podmínkou pro efektivní použití MKP je automatizace tvorby sítě konečných prvků. V průběhu vývoje měly pokusy o automatizaci různé formy. Dnes je nejrozšířenějším přístupem generování MKP sítí právě do

geometrických modelů. Podstatou tohoto přístupu je využití geometrického modelu (jehož tvorba je podstatně méně pracná než-li ruční tvorba sítě) jako šablony, s jejíž pomocí vygeneruje automatický generátor sítě množiny uzlů a elementů, které tvoří MKP model tělesa. Pro názornost a zjednodušení výpočtů byla pro řešení úloh zvolena plošná úloha.

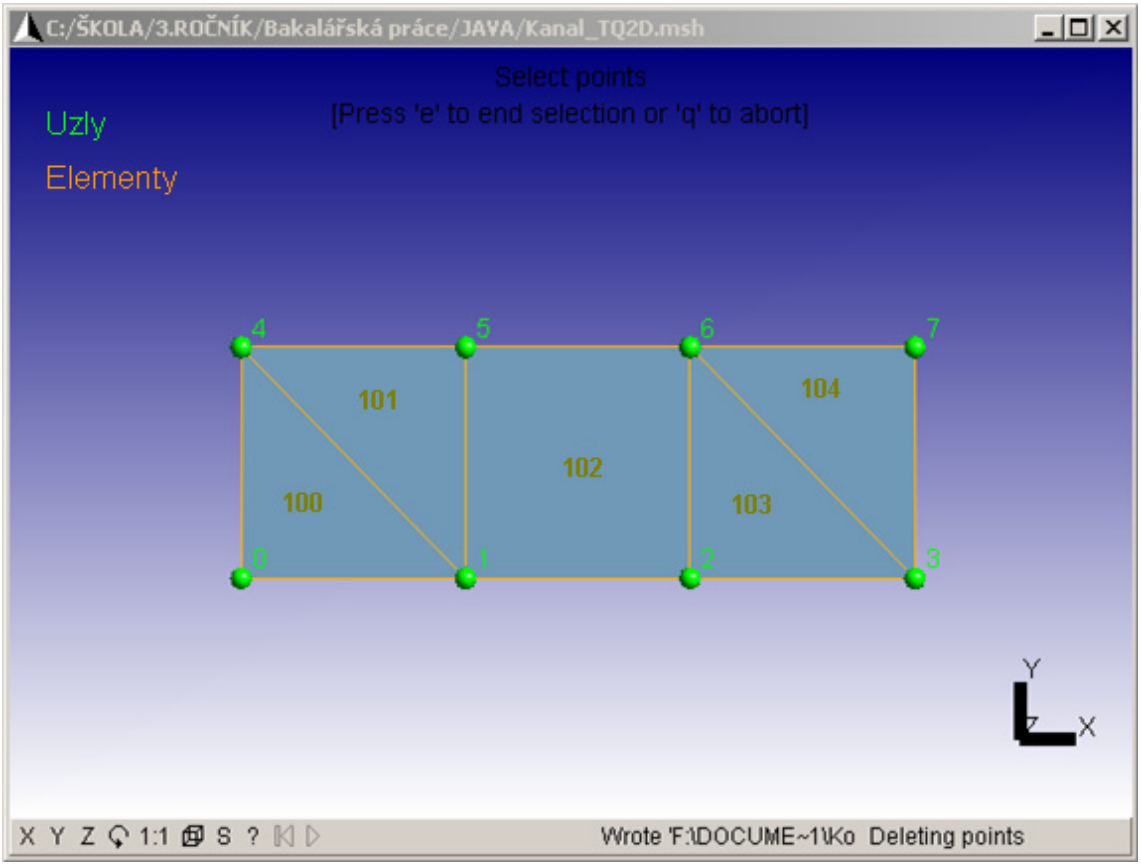
5 Řešení v jazyce Java

Pro vytvoření seznamů entit potřebných pro práci v síti se nejlépe hodí právě tento programovací jazyk. Pomocí jednoduchých základních příkazů lze vytvořit přehledný programovací kód. Výhodou je dokonalá datová kontrola oproti jiným programovacím jazykům. Pracuje se s tzv. třídami, ve kterých se mohou programovat jednoduché algoritmy a ty poté nezávisle testovat. Následující části tohoto textu se zprvu zaměřují na teoretické předpoklady, které jsou zde prezentovány do tabulek (seznamů) a posléze je připravena praktická ukázka v Jave. Řešení je možno naprogramovat různými způsoby. Nejlepší možná realizace je pomocí dynamických polí a řetězců.

Při řešení úloh se vycházelo z literatury [1] Ing. Dalibora Frydrycha, Ph.D. Využilo se především základních technik a metod pro práci v jazyce Java. Základní zdrojový kód řešil jednoduché problematiky uzlů, například kolik si daný uzel drží elementů, jaké má souřadnice, jaké má sousedy přes element, dále pak výpis elementů a jakými je konkrétní element tvořen uzly. Tyto základy byly využity k výstavbě kompletního popisu síťových prvků a to například k získání sousedních elementů, které s daným elementem sousedí přes uzel, celkový výpis hran i s globálním a lokálním označením. Uzly, které danou hranu tvoří. Dále také jaký si daná hrana drží element a samozřejmě naopak - jaké hrany si celkově drží daný element. Nejobtížnější byla část, kdy se řešila otázka, zda daná hrana sousedí i s jinou hranou a pokud ano, tak obecně vytvořit aktivní vyhledávání sousedních elementů přes společné hrany. Pracuje se především s těmito třídami: MeshTest2.java, Mesh.java, Element.java, Line.java. Vzniklo zde najednou mnoho možných kombinaci entit a jejich možné topologie, proto se níže tato práce věnuje jejich podrobnějšímu popisu.

5.1. Rozbor jednoduché sítě

Na následujícím obrázku je zobrazena jednoduchá síť s elementy jak trojúhelníkovými tak i jedním čtyřúhelníkem. S touto sítí se pracuje v celé práci, jelikož je vhodná jako názorný příklad. Uzly jsou označeny hodnotami , které jsou náhodně přiřazené programem. Taktéž je to i u elementů. Může si povšimnout určitých zákonitostí, které se v modelu a jeho topologii objevují a které jsou z hlediska této práce důležité.



Obr.3 *Základní model zkoumané oblasti*

Předešlý obrázek je vytvořen v programu Gmsh a simuluje například chování proudění kapaliny v podzemí, šíření teploty materiálem, obecně si pod tím můžeme představit popis skoro jakéhokoliv fyzikálního jevu vzniklého na libovolném prostoru nebo předmětu. Je patrné, že uzly a elementy jsou spolu spjaty. Jeden bez druhého nemohou existovat. Vždy, když se vytváří nový element, musí se nejprve definovat pole, seznam uzlů a poté přiřadit dané uzly elementu.

5.1.1. Vkládání Uzlů a Elementů

Teoretický předpoklad může vypadat tak, jak je tomu na následujícím seznamu. Vytvoříme si seznam jak elementů tak uzlů a vypíšeme si jej tak, aby se dané prvky ukládali a řadili do posloupností. Zajímají nás označení těchto prvků a jejich celkový počet.

Seznam 1 ***SLOŽENÍ SÍTĚ***

Elementy	100	101	102	103	104			
Uzly	0	1	2	3	4	5	6	7

V Jave si tento seznam vytvoříme dynamicky ve třídě Mesh.java. Tato třída je rodičem[2] třídy MeshTest2.java:

```
public class MeshTest2 extends Mesh {
}
```

Přidání uzlů do sítě:

Vytvoříme si seznam uzlů(může být náhodný) ve třídě MeshTest2.java. V následující části si načteme 4 uzly s jejich názvy a souřadnicemi:

```
public MeshTest2() {
        this(1000,1000,1000); // voláme jiný konstruktor stejné třídy
                              // nastavení pole Uzel, Element, Hrana
    }
```

Způsob vložení jednotlivých uzlů do sítě kde se definuje typ uzlu jako plošný prvek, název uzlu a jeho souřadnice:

```
    public void test() throws MeshException {
        addNode( new Node2D( 0, 0.0, 0.0 ) );   // Vytvoření 4 Uzlů
        addNode( new Node2D( 1, 1.0, 0.0 ) );   // (nazev, souřadnice)
        addNode( new Node2D( 2, 2.0, 0.0 ) );
        addNode( new Node2D( 4, 0.0, 1.0 ) );
```

[2] Nová třída tedy dědí všechny vlastnosti svého rodiče.

Pokud bychom do seznamu zařadili uzel se stejným označením, který již v poli je, vyvolá se v programu výjimka. Je to důležité z hlediska jedinečnosti každého prvku v síti.

Přidání dalšího uzlu a vyvolání výjimky:

```
        addNode( new Node3D( 8, 3.0, 1.0, 1.0 ) );
// výstupní konzole:
Two nodes with the same label : 8
```

Problém může vzniknout i tehdy, kdybychom do pole chtěli vložit uzel o jiné dimenzi nežli dva - například trojdimenzionální souřadnice:

```
      addNode( new Node3D( 9, 3.0, 1.0, 1.0 ) );
// výstupní konzole:
New node with label 9 has dimension 3
But in mesh are nodes with dimension 2
```

Máme tedy vybrané uzly a nyní bychom je chtěli vypsat na výstupní konzoli. To znamená načíst je z pole. Pole je vytvořeno takto:

```
    protected int numNodes = 0;
    protected Node[ ] nodes;
```

Pokud je pole malé a počet uzlů je větší než jeho velikost, vytvoříme nové pole, do kterého se zkopíruje obsah původního a jeho velikost je dvojnásobná. Posléze automaticky **garbage collector** původní pole smaže, jelikož je nepotřebné a zbytečně zabírá místo v paměti:

```
    protected void extendNodes() {
        Node[ ] newNodes = new Node[ 2*nodes.length ];
        System.arraycopy( nodes, 0, newNodes, 0, nodes.length );
        nodes = newNodes;
```

Vytvoříme si metody getIndex a getLabel. Pro každý uzel byla ještě vytvořena metoda Indexace, což se využilo i v práci s elementy a především později v indexaci hran:

```
    public int getIndex() {
```

```
        return index;
    }
     public int getLabel() {
        return label;
    }
```

Metoda pro přiřazení názvu (labelu) uzlu:

```
    public int getIdxNodeLabel( int label ) throws MeshException {
        for ( int in = 0; in < numNodes; ++in ) {
            if( nodes[ in ].getLabel() == label ) return in;
        }
        throw new MeshException( "Mesh.getIdxNodeLabel()\n" + "Unknown
node with label " + label );
    }
```

Metoda pro přiřazení indexu uzlu:

```
public int getLabelNodeIdx( int in ) throws MeshException {
        if( in >= numNodes ) throw new MeshException(
"Mesh.getLabelNodeIdx\n" + "Unknown node with index " + in );
        return nodes[ in ].getLabel();
    }
```

Metoda pro vrácení souřadnice uzlu:

```
     public double getCoor( int i ) {
        return coor[ i ];
    }
```

Metoda getNumNodes nám vrací proměnnou numNodes, která nám definuje počet uzlů:

```
    public int getNumNodes() {
        return numNodes;
    }
```

Díky těmto základním a dalším pomocným třídám, které ale z hlediska zdlouhavosti náročnosti kódu nejsou uvedeny, je možno vytvořit postupné načítání uzlů do pole a posléze jejich výpis na konzoli s jejich názvem, indexem a souřadnicí.

Výstup na konzoli:

```
NODE
****
NodesLabel(Index),Coor(x,y)
----------------------------
 0(0,0)    0.0  0.0
 1(1,1)    1.0  0.0
 2(2,2)    2.0  0.0
 3(3,3)    3.0  0.0
 4(4,4)    0.0  1.0
 5(5,5)    1.0  1.0
.
.
```

Přidání elementů do sítě:

V předešlé části jsme se věnovali systému vkládání uzlů. Bez vytvořených uzlů nelze sestavit dané elementy.

Vytvoření polí pro jednotlivý elementy:

```
int[] nodes0 = { 0, 1, 4 };
int[] nodes1 = { 5, 4, 1 };
int[] nodes23 ={ 1, 2, 6, 5 };
```

Systém vložení jednotlivých elementů do sítě, kde se definuje typ elementu(Triangle, Quadrangle) a v závorce je uvedeno označení elementu, pole uzlů, které jej tvoří a region, do kterého daný element spadá:

```
addElement( new Triangle( 100, nodes0, regions ) );
addElement( new Triangle( 101, nodes1, regions ) );
addElement( new Quadrangle( 102, nodes23, regions ) );
```

Přidávání elementů podle daných uzlů- způsob pamatování si uzlů v elementu:

```
public void addElement( Element element ) throws MeshException {

        if ( element.getNodeInx( 0 ) == null ) {
            for ( int iin = 0; iin < element.getNumNodes(); ++iin ) {
                int label = element.getLabelNodeInx( iin );
                Node node = getNodeLabel( label );
```

```
                element.setNodeInx( node, iin );
            }
        }
        if ( isElementLabel( element.getLabel() ) ) {
            throw new MeshException( "Mesh.addElement()\n" + "Two
elements with the same label : " + element.getLabel() );
        }
```

Jako v případě vytváření pole uzlů i zde se vytvořilo pole elementů, také bylo potřeba nadefinovat metody, které zajistí výpis elementů. Nebudeme se jimi ale již blíže zabývat, jelikož tyto metody jsou vytvořeny stejným způsobem jako u uzlů.

5.1.2. Vztah uzel-element

Pro orientaci v síti je jedno z nejdůležitějších znát počet a názvy elementů, které jsou na konkrétním uzlu „přichyceny“. Každý uzel si tedy pamatuje seznam elementů, se kterými je svázán viz Seznam 2 a naopak viz Seznam 3 .

Seznam 2 ***UZLY X ELEMENTY***

Uzly	**Připojené elementy**		
uzel 0	100		
uzel 1	100	101	102
uzel 2	102	103	
uzel 3	103	104	
uzel 4	100	101	
uzel 5	101	102	
uzel 6	102	103	104
uzel 7	104		

Zde se vypisuje ze seznamu počet elementů navázaných na uzlu a jejich název:

```
System.out.print( "[" + node.getNumElements() + "]" );
       for ( int iie = 0; iie < node.getNumElements(); ++iie ) {
System.out.print( "  " + elements[ node.getIdxElementInx( iie )
].getLabel() );
     }
```

Výstup na konzoli:

```
NODE
****
NodesLabel,(Index)
[]- Number of Elements neighbouring the Node , Labels of this Elements
-----------
0(0,0)
 [1]  100
1(1,1)
 [3]  100  101  102
2(2,2)
 [2]  102  103
3(3,3)
 [2]  103  104
4(4,4)
 [2]  100  101
.
.
```

5.1.3. Vztah element-uzel

Každý element si musí pamatovat seznam uzlů které ho tvoří.

Seznam 3 ***ELEMENTY X UZLY***

Eementy	**Připojené uzly**			
100	uzel 0	uzel 1	uzel 4	
101	uzel 1	uzel 4	uzel 5	
102	uzel 1	uzel 2	uzel 5	uzel 6
103	uzel 2	uzel 3	uzel 6	
104	uzel 3	uzel 6	uzel 7	

V této části se ze seznamu vypisuje počet uzlů navázaných na daném elementu a jejich název:

```
      for ( int ie = 0; ie < getNumElements(); ++ie ) {
            elm = (Element)elements[ ie ];
            if ( elm != null ) {
System.out.print( "  " + elm.getLabel() + "\t(" + elm.getIndex() + ","
+ ie + ")" + "\t\t nNodes: " + elm.getNumNodes() + "    " );
       for ( int iin = 0; iin < elm.getNumNodes(); ++iin ) {
             Node node = elm.getNodeInx( iin );
System.out.print("\t  "+ node.getLabel() + "(" + node.getIndex() +
")");
   };  }
 }
```

Výstup na konzoli:

```
ELEMENT
*******
ElementLabel,(Index), Number of nodes, Label(index)Nodes generating
Element
-----------
  100 (0,0)        nNodes: 3          0(0)        1(1)        4(4)
.
.
```

5.1.4. Vztah uzel-uzel

Každý uzel si také musí pamatovat, které sousedy má přes elementy okolo sebe. Nepotřebuje si pamatovat sám sebe.

Seznam 4 ***UZLY X UZLY (SPOLEČNÝ ELEMENT)***

Uzly	**Sousední uzly**					**Přes elementy:**
uzel 0	uzel 1	uzel 4				**100**
uzel 1	uzel 0	uzel 2	uzel 4	uzel 5	uzel 6	**100, 101, 102**
uzel 2	uzel 10	uzel 3	uzel 5	uzel 6		**102, 103**
uzel 3	uzel 2	uzel 6				**103, 104**
uzel 4	uzel 0	uzel 1	uzel 5			**100, 101**
uzel 5	uzel 1	uzel 2	uzel 4	uzel 6		**101, 102**
uzel 6	uzel 1	uzel 2	uzel 3	uzel 5	uzel 7	**102, 103, 104**
uzel 7	uzel 3	uzel 6				**105**

Metody pro vytvoření sousedních uzlů:

```
  public int getNumNeighbours() {
        return sNbrIdx.size();
    }
          public void addNeighbourIdx( int inn ) {
        int idxInt = Collections.binarySearch( sNbrIdx, inn );
        if ( idxInt < 0 ) sNbrIdx.add( -( idxInt +1 ), inn );
    }
```

Syntaxe pro vypisování sousedních prvků:

```
System.out.print( "[" + node.getNumNeighbours() + "]" );
       for ( int inn = 0; inn < node.getNumNeighbours(); ++inn ) {
System.out.print( "  " + getNodeIdx( node.getIdxNeighbourInx( inn )
).getLabel());
 }
```

Výstup na konzoli:

```
NODE
****
NodesLabel,(Index)
 []- Number of Nodes neighbouring the Node over the Elements ,  Labels
of this Nodes
--------
0(0,0)
 [2]  1  4
1(1,1)
 [5]  0  2  4  5  6
2(2,2)
 [4]  1  3  5  6
3(3,3)
 [3]  2  6  7
4(4,4)
 [3]  0  1  5
5(5,5)
 [4]  1  2  4  6
.
.
```

5.1.5. Vztah element-element

Následující úvaha popisuje, že každý element si dokáže zjistit, s jakým elementem sousedí přes svůj uzel.Zajímavé je, že své sousedy nemají pouze přes krajní uzly sítě. Stejně jako u uzlů i u elementu není potřeba a spíše to je i nevhodné, aby si daný element pamatoval sám sebe.

Seznam 5 ***ELEMENTY X ELEMENTY (SPOLEČNÝ UZEL)***

	Sousední elementy				**Přes uzly:**
element 100	element 101	element 102			**1, 4**
element 101	element 100	element 102			**1, 4, 5**
element 102	element 100	element 101	element 103	element 104	**1, 2, 5, 6**
element 103	element 102	element 104			**2,3,6**
element 104	element 102	element 103			**3, 6**

Metoda getElementsNeighbourOverNodes vrací přímo text, který obsahuje seznam sousedních elementů sousedících s elementem přes uzel:

```
private String getElementsNeighbourOverNodes(Element element) {
        String pom = "";
        int countElements = 0;
        int tab = 0;
        int elementLabel = element.getLabel();
```

Tato část prochází všechny uzly elementu z parametru:

```
        Node[] pomNodes = element.getNodes();
        for (int iin=0;iin<element.getNumNodes();iin++) {
            Node node = pomNodes[iin];
   String elementOverOneNode = "  *\t node("+node.getLabel()+")=";
```

Zde funkce prochází všechny elementy v síti:

```
            for (int jjn=0;jjn<numElements;jjn++) {
                Element element2 = this.elements[jjn];
```

Následující řádek eliminuje situaci, kdy uvažujeme dva totožné elementy:

```
                if (element2.getLabel()==element.getLabel()) continue;
```

Další část volá metodu hasnode, která testuje, zda-li je uzel součástí pole uzlů prvního parametru, tj. elementu2.getNodes(), pokud ano, tak spolu sousedí, jinak ne:

```
                if (this.hasNode(element2.getNodes(),node)) {
                    countElements++;
                    elementOverOneNode += " "+element2.getLabel();}
            }
          pom += elementOverOneNode;    }
        return pom; }
```

Testuje, jestli vložené pole obsahuje uzel, vrací true nebo false:

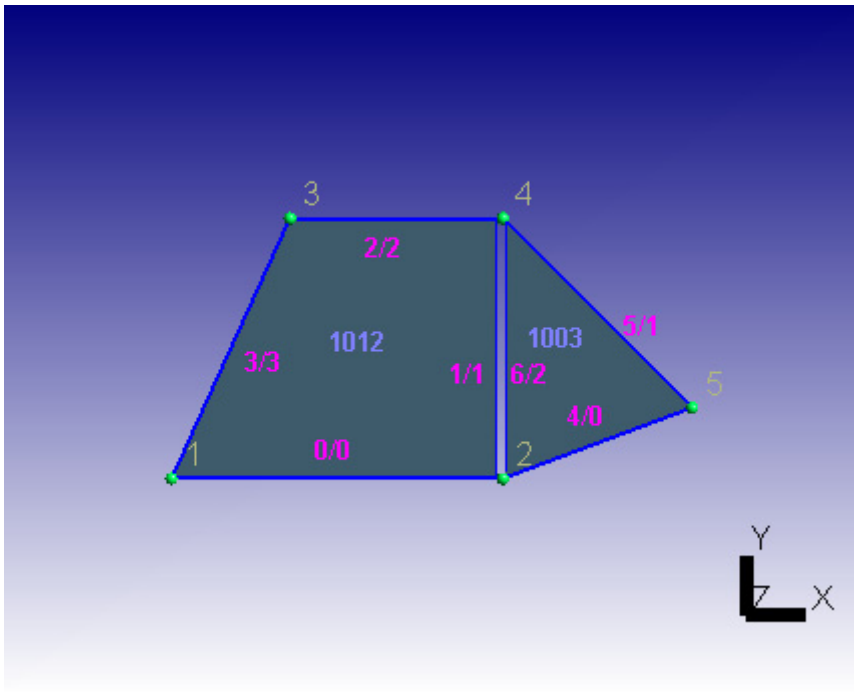
```
private boolean hasNode(Node[] pomNodes,Node pomNode) {
        for (int iin=0;iin<pomNodes.length;iin++) {
            if (pomNode.getLabel()==pomNodes[iin].getLabel()) {
                return true; }
        }
        return false;    }
```

Výstup na konzoli:

```
ELEMENT
*******
ElementLabel,(Index),
 *  Labels Elements over each Node
--------
101   (1,1)
 *node(5)= 102     *node(4)= 100    *node(1)= 100 102
............
102   (2,2)
  *node(1)= 100 101  *node(2)= 103  *node(6)= 103 104  *node(5)= 101
.
.
```

5.2. Problematika hran

Z následujícího obrázku je lehce možno vyčíst, že pokud danou oblast tvoří například dva elementy 1012 a 1003, jsou k sobě „upoutány“ přes svoje hrany.



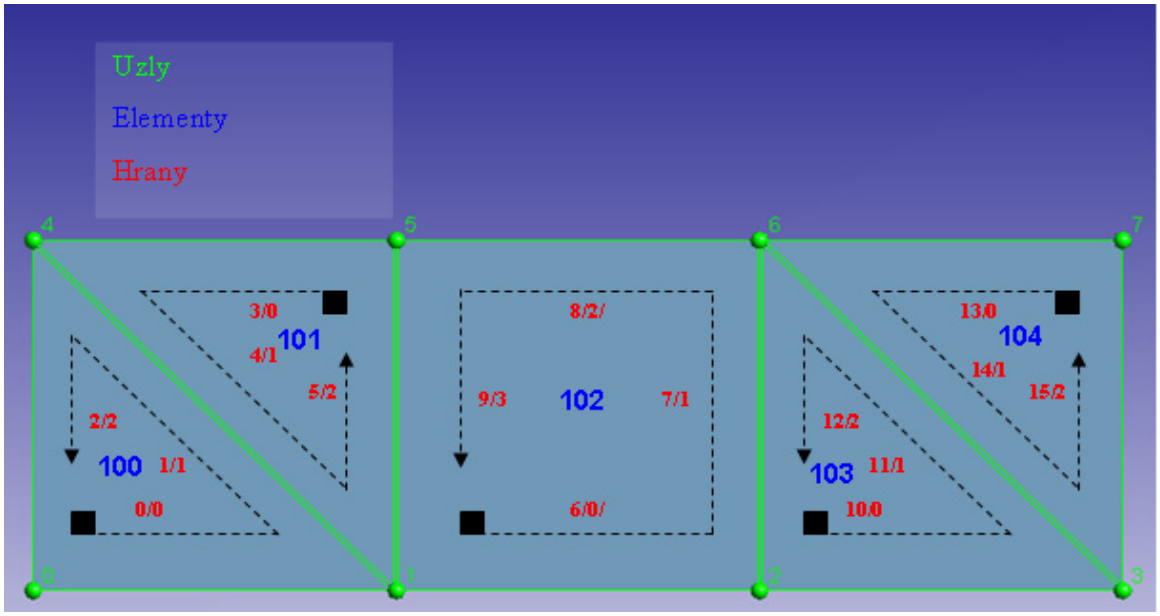
Obr.4 *Rozdělení hran*

Pro názornost vypadají elementy odděleně, ve skutečnosti tomu vizuálně tak samozřejmě není, ale pro demonstraci je to vhodné. Aby se určily zákonitosti v označení, dostali hrany elementů svoje jak lokální , tak globální označení, přičemž globální je důležitější, jelikož pomocí této hodnoty (tohoto labelu) program vyhledává spojitosti s uzly a elementy.

Dva vzorové elementy (1012 a 1003) mají tedy společné hrany 1/1 a 6/2. První číslice před lomítkem znamená globální název, druhé číslo lokální (vnitřní).

5.2.1. Vztah hrana-uzel

V první řadě je potřeba znát. Kolik je vlastně v dané síti hran. Pro hrany je opět v Jave vytvořeno nové pole hran, které se při nedostatku místa v poli zdvojnásobí. Důležité je si uvědomit, že každou hranu v síti tvoří vždy 2 uzly.



Obr.5 *Rozšířený model zkoumané oblasti o popis hran*

Seznam 6 ***HRANY***

Globální označení	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Lokální označení	0	1	2	0	1	2	0	1	2	3	0	1	2	0	1	2

Zaměříme se na síť na Obr.5 . Teoreticky na ní můžeme spočítat celkem 16 hran (0-15). V jazyce Java je toto ošetřeno obecně pro libovolný počet hran. Nejprve se přidává hrana mezi body elementu, tj. mezi uzly. Načítání je jednoduché. Začíná se vkládat hrana od prvního uzle po poslední a po projetí celého elementu se ještě jinou metodou uzavře poslední uzel s prvním. Poté se pokračuje na dalším elementu, ve stejném cyklu.

Můžeme si všimnout, že některé hrany jsou tvořeny dvěma stejnými uzly. To pro nás znamená, že spolu sousedí a určují nám souseda přes hranu.

Seznam 7	***HRANY X UZLY***	
Hrany		
Globální / Lokální	**Uzly hrany**	
0 / 0	uzel 0	uzel 1
1 / 1	uzel 1	uzel 4
2 / 2	uzel 0	uzel 4
3 / 0	uzel 4	uzel 5
4 / 1	uzel 1	uzel 4
5 / 2	uzel 1	uzel 5
6 / 0	uzel 1	uzel 2
7 / 1	uzel 2	uzel 6
8 / 2	uzel 5	uzel 6
9 / 3	uzel 1	uzel 5
10 / 0	uzel 2	uzel 3
11 / 1	uzel 3	uzel 6
12 / 2	uzel 2	uzel 6
13 / 0	uzel 6	uzel 7
14 / 1	uzel 3	uzel 6
15 / 2	uzel 3	uzel 7

Tato metoda probíhá v cyklu for, aby se určily všechny hrany:

```
            if (firstNodeForLine==null) {
                firstNodeForLine = node;
            } else {
```

Tato část kódu rozšiřuje pole hrany, pokud je daná hrana předposlední. To má význam takový, že se hrany počítají jinak než uzly a mohlo by dojít k chybě:

```
if ( (numLines == (lines.length-1)) || (numLines == lines.length))
extendLines();
```

Zde je dokončení přiřazování uzlů hranám, jejich region a název elementu, který tuto hranu tvoří:

```
this.lines[numLines] = new Line(numLines, new Node[]
{firstNodeForLine,node},element.getRegions(),iin-1,
element.getLabel());
                ++numLines;
                firstNodeForLine = node;
            }
```

Nutností je uzavřít hranou poslední a první uzel:

```
 Node lastNodeForLine = getNodeLabel(element.getLabelNodeInx(0));
this.lines[numLines] = new Line(numLines, new Node[]
{firstNodeForLine,lastNodeForLine},element.getRegions(),element.getNum
Nodes()-1,element.getLabel());
```

Metoda , která zajistí výpis názvu a indexu uzlů na konzoli:

```
    private String printNodes(Line line) {
        String pom = "";
        for ( int iin = 0; iin < line.getNumNodes(); iin++ ) {
            Node node = line.getNodeInx(iin);
            pom += node.getLabel() + "(" + node.getIndex() + ") ";
        }
        return pom;
    }
```

Výstup na konzoli:

```
LINES
*****
Line(Glob,Loc),Label(index)Nodes generating line
--------------------------------
  (0,0)       nNodes: 2  0(0) 1(1)
  (1,1)       nNodes: 2  1(1) 4(4)
  (2,2)       nNodes: 2  4(4) 0(0)
  (3,0)       nNodes: 2  5(5) 4(4)
  (4,1)       nNodes: 2  4(4) 1(1)
  (5,2)       nNodes: 2  1(1) 5(5)
.
.
```

5.2.2. Vztah hrana-element

Je potřeba si dále určit, která hrana má jaký element. Z předešlé znalosti hrana-uzly již není problém to určit podle názvu uzlů.

Vrací nám hodnotu–název elementu:

```
    public int getParentElementIdx() {
        return this.parentElementIdx; }
```

Seznam 8	***HRANA X ELEMENT***								
Hrany Globální / Lokální	0/0	1/1	2/2	3/0	4/1	5/2	6/0	7/1	8/2
Element	100	100	100	101	101	101	102	102	102
	9/3	10/0	11/1	12/2	13/0	14/1	15/2	9/3	10/0
	102	103	103	103	104	104	104	102	103

Výstup na konzoli:

```
LINES
*****
Line(Glob,Loc), LabelElement generating line
--------
  (0,0)                    [100]
  (1,1)                    [100]
  (2,2)                    [100]
  (3,0)                    [101]
  (4,1)                    [101]
.
.
```

5.2.3. Vztah element-hrany

Na výpisu v konzoli je důležité, pro pozdější vyhledávání, znát strany, které tvoří element. Zajímá nás především počet hran a jejich název.

Seznam 9	***ELEMENT X HRANY***			
Eement	**Hrany elementu**			
100	0/0	1/1	2/2	
101	3/0	4/1	5/2	
102	6/0	7/1	8/2	9/3
103	10/0	11/1	12/2	
104	13/0	14/1	15/2	

Realizace zdrojového kódu, kde se hledají hrany elementu, vypadá takto:

```
private String getLinesNameOfElement(Element element) {
        String pom = "";
        int countLines = 0;
        int elementLabel = element.getLabel();
        for (int iin=0;iin<numLines;iin++) {
            Line line2 = this.lines[iin];
```

Test zda-li hrana je součástí vybraného elementu:

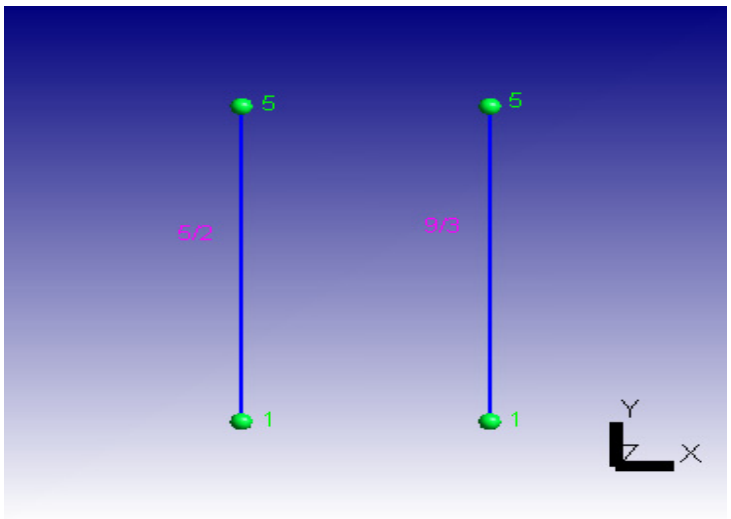
```
            if (elementLabel==line2.getParentElementIdx()) {
                pom += "    "+line2.getLabel()+"/"+line2.getIndex();
                countLines++;       }
        }
        return "["+countLines+"]"+pom; }
```

Výstup na konzoli:

```
ELEMENT
*******
ElementLabel,(Index),
[]- Number of lines generating Element , Labels of this Lines
-------------------------------
  100 (0,0)
  [3]    0/0    1/1    2/2
............
  101 (1,1)
  [3]    3/0    4/1    5/2
.
.
```

5.2.4. Vztah hrana-hrana

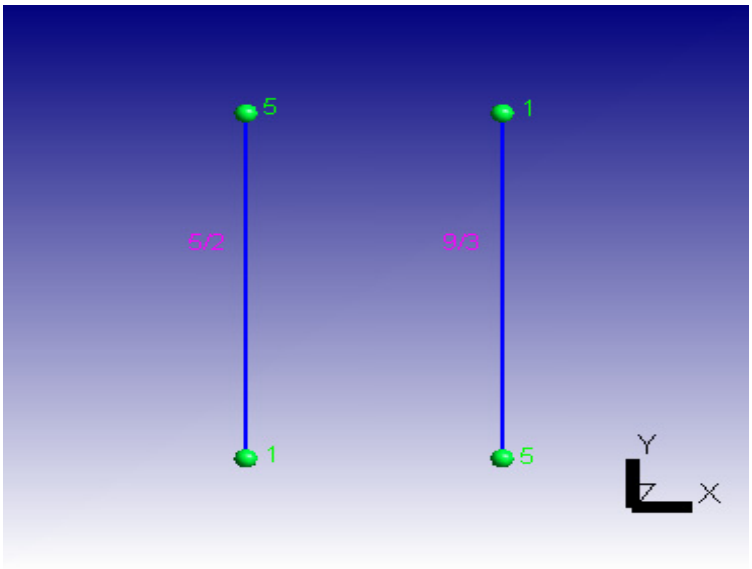
Jak již bylo zmíněno v předešlé části této práce, každá hrana má dva uzly, které tvoří daný element, nebo elementy. Předmětem tohoto zkoumání je možnost, že dva uzly si drží dvě totožné hrany s různými názvy, které jsou vedle sebe, tj.spolu přímo sousedí. V programu se především řeší situace porovnávání hran podle názvu uzlů, které si je drží. Na následujícím obrázku je vidět, že hrana 5/2 sousedí s hranou 9/3 a mají společné uzly [1-5 a 1-5].



Obr.6 *Hrany ve stejném uspořádání*

Tento přiklad je vybrán z rozšířeného modelu (Obr.5 aby bylo možno přesně demonstrovat praktický problém. Jedná se o dva elementy 101 a 102, které spolu sousedí přes dvě hrany. Pokud cyklus načte hrany v pořadí uzlů [1-5 a 1-5] stačí nastavit podmínku značně jednoduše.

V programu ale může nastat situace, kdy při porovnávání hran se sousední hrana načte s opačným pořadím uzlů. [1-5 a 5-1] To je nutno ošetřit podmínkou složitější viz Obr.7



Obr.7 *Hrany v opačném uspořádání*

Daná část následujícího kódu řeší postupně všechny nutné podmínky pro konečný výpis na konzoli:

```
    private Line getNeighbourLine(Line line) {
        Line line2 = null;
```

Dále vrací názvy obou uzlů hrany, metoda getNodes vrací pole, z něj chceme první uzel (0) a z něho jeho název (getLabel()):

```
        int node1Label1 = line.getNodes()[0].getLabel();
        int node1Label2 = line.getNodes()[1].getLabel();

        for (int iin=0;iin<numLines;iin++) {
            line2 = lines[iin];
            if (line.getLabel()==line2.getLabel()) continue;
```

```
            int node2Label1 = line2.getNodes()[0].getLabel();
            int node2Label2 = line2.getNodes()[1].getLabel();
```

Již zmíněná podmínka pro porovnání stejných a opačných hran je realizována v následujícím textu.

```
if ((node1Label1==node2Label1) && (node1Label2==node2Label2) ||
(node1Label2==node2Label1) && (node1Label1==node2Label2)) {
                return line2;
            }
```

Výstupní metoda pro konzoli pak vypadá takto.

```
    private String getNeighbourName(Line line) {
        if (line!=null) {
            return "\t("+line.getLabel()+"/"+line.getIndex()+")";
        } else {
            return "\tNo neighbour!";
        }
    }
```

Výstup na konzoli:

```
LINES
*****
Line(Glob,Loc)  Neighbour Line

------------------------------
  (0,0)     No neighbour!
  (1,1)     (4/1)
  (2,2)     No neighbour!
  (3,0)     No neighbour!
  (4,1)     (1/1)
  (5,2)     (9/3)
.
.
```

5.2.5. Vztah element-element

K vyřešení této problematiky bylo potřeba zjistit většinu předešlých informací ze sítě. Proto se jí tento text zabývá až skoro nakonec. Z poznatku sousedících hran již nyní nic nebrání tomu, sestavit přehled sousedících elementů právě přes již zmíněné hrany. Jakýkoliv element v plošné úloze diskretizačních sítí musí vždy zákonitě mít alespoň jeden sousední element.

Seznam 10 ***ELEMENTY X ELEMENTY (PŘES HRANY)***

Elementy	**Sousední elementy**	
element 100	element 101	
element 101	element 100	element 102
element 102	element 101	element 103
element 103	element 102	element 104
element 104	element 103	

Programově je to ošetřeno tak, že nejprve se prochází hrany sítě, poté hledá element k dané hraně, vyhledá sousední hranu, pokud existuje, existuje element:

```
private String getElementsNeighbourOverLines(Element element) {
        String pom = "";
        int countElements = 0;
        int elementLabel = element.getLabel();
```

Cyklus for prochází všechny hrany sítě:

```
        for (int iin=0;iin<numLines;iin++) {
            Line line2 = this.lines[iin];
```

Test zda-li hrana je součástí vybraného elementu:

```
            if (elementLabel==line2.getParentElementIdx()) {
```

Vrací sousední hranu, pokud existuje, jinak vrací null:

```
                Line pomLine = this.getNeighbourLine(line2);
```

Pokud existuje, tak ji jako všude připojí k výslednému řetězci:

```
                if (pomLine!=null) {
                    countElements++;
                   pom+= "    " + pomLine.getParentElementIdx();}
            }
        }
        return "["+countElements+"]"+pom;
    }
```

Výstup na konzoli:

```
ELEMENT
*******
ElementLabel,(Index),
 []- Number of Elements neighbouring with Element over the Lines ,
 Labels of this Elements
--------
100(0,0)
  [1]    101
............
101(1,1)
  [2]    100    102
..
```

6 Závěr

Tato práce se snažila nalézt obecná řešení a úvahy, které by napomohly k vytváření a práci s prvky v diskretizačních modelech. V úvodním textu se řeší problematika MKP a návrh sítí v programu Gmsh. Po získání základních dovedností pro práci s objekty v jazyce Java se přistoupilo k nejdůležitější části - vytváření tříd a v nich metod, které pracovaly s entitami modelu. Jelikož je tato problematika poměrně rozsáhlá, zaměřili jsem se na oblast nalezení vztahů mezi elementy, uzly a hranami, doplňování sítě o nové prvky a použití základních datových struktur pro jejich uložení. Povedlo se vytvořit podle předem zjištěných předpokladů funkční metody, jež vedly k nalezení vztahů mezi různými entitami.

Jeden z největších problémů bylo vyhledávání a orientace elementů v síti vůči jiným prvkům přes sousední hrany, což se nakonec podařilo dořešit. V poslední části se pozornost věnovala testování většiny metod na vhodných příkladech. Zadání se tímto podařilo splnit.

Největší výhoda tohoto řešení spočívá v možnosti načtení libovolně velké sítě a pomocí jednoduchých vstupních úkonů automaticky vytvořit ucelený a přehledný seznam všech společných prvků a entit s jejich detailním popisem.

Literatura a ostatní použité zdroje

[1] FRYDRYCH D., *Metodika implementace metody konečných prvků DF²EM [on line]*. 2007, 2. kv2tna 2007

[2] ZIENKIEWICZ O.C, TAILOR R.L.: *Finite Element Method* (5th Eition) *Volume 1 – The basis*, Elsevier, 2000, ISBN 0-7506-5049-4

[3] KOLÁŘ V., NĚMEC I., KANICKÝ V.: FEM *Principy a praxe metody konečných prvků*, Computer Press, ISBN 80-7226-021-9

[4] HEROUT P.: *Učebnice jazyka Java*, Kopp, ISBN 80-7232-115-3

[5] HAWLITZEK F.: *Java 2-příručka programátora*, Grada Publishing, ISBN 80-247-9060-2

[6] BRŮHA L.: *Java-Hotová řešení*, Computer Press, ISBN 80-251-0072-3

[7] KOTALA Z., TOMAN P., *JAVA, sborník,* Západočeská univerzita (1997-2001)

[8] ŠEMBERA J., FRYDRYCH D.: *Stavba a řešení počítačových modelů* [online], skriptum, 10. března 2007, Liberec 2003, 2004

[9] Dokumentace jazyka Java: Sun Microsystems, [online], 15. ledna 2007, http://www.sun.com/

[10] JIRÁNEK P., *Výukové texty*, Technická univerzita Liberec, Katedra modelování, http://flow.kmo.tul.cz/~www/czech/vyuka.php

[11] GEUZAINE CH., REMACLE J.-F.: *A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities* [online], 12. března 2007, Version 2.0, February 5 2006, http://www.geuz.org/gmsh/

[12] MACUR J.: *Dynamické systémy*, Vysoké učení technické v Brně, *Vyukové materiály kapitola 7* [online], 3. ledna 2007, AIU-Fest, http://www.fce.vutbr.cz/studium/materialy/Dynsys/kap7/kap7.htm

[13] Otevřená internetová encyklopedie, *Wikipedia* [online], poslední editace 9.května 2007, http://cs.wikipedia.org

[14] BOUCNIK P., *S2S-Science to Science* [online], 10.listopadu 2006, http://www.boucnik.cz/kap6.htm

Příloha A:

Vstupní hodnoty z jazyka Java pro vytvoření diskretizační sítě pro tvorbu vztahů různých entit.

Využily se přímo tyto třídy: MeshTest2.java, Mesh.java, Element.java, Line.java

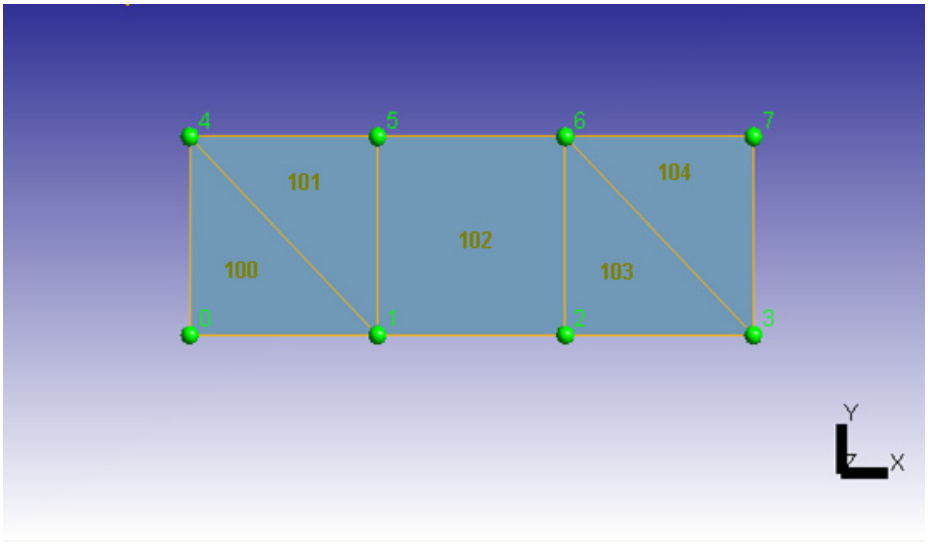
```
        addNode( new Node2D( 0, 0.0, 0.0 ) );
        addNode( new Node2D( 1, 1.0, 0.0 ) );
        addNode( new Node2D( 2, 2.0, 0.0 ) );
        addNode( new Node2D( 3, 3.0, 0.0 ) );

        addNode( new Node2D( 4, 0.0, 1.0 ) );
        addNode( new Node2D( 5, 1.0, 1.0 ) );
        addNode( new Node2D( 6, 2.0, 1.0 ) );
        addNode( new Node2D( 7, 3.0, 1.0 ) );

        int[] regions = { 0 };
        int[] nodes0 = { 0, 1, 4 };
        int[] nodes1 = { 5, 4, 1 };
        int[] nodes23 ={ 1, 2, 6, 5 };
        int[] nodes4 = { 2, 3, 6 };
        int[] nodes5 = { 7, 6, 3 };

        addElement( new Triangle( 100, nodes0, regions ) );
        addElement( new Triangle( 101, nodes1, regions ) );
        addElement( new Quadrangle( 102, nodes23, regions ) );
        addElement( new Triangle( 103, nodes4, regions ) );
        addElement( new Triangle( 104, nodes5, regions ) );
```

V programu Gmsh vypadá modelovaná oblast takto:



Výstupní konzole v jazyce Java:

```
MeshTest2.main() - Start

Width=3.0
Height=1.0

NODE
****
Number of all Nodes=8      Array of Nodes[1000]

Comments:
--------
NodesLabel,(Index),Coor(x,y)
[]- Number of Elements neighbouring the Node , Labels of this Elements
[]- Number of Nodes neighbouring the Node over the Elements ,  Labels
of this Nodes
--------
 0(0,0)   0.0  0.0
[1]  100
[2]  1  4

 1(1,1)   1.0  0.0
[3]  100  101  102
[5]  0  2  4  5  6

 2(2,2)   2.0  0.0
[2]  102  103
[4]  1  3  5  6

 3(3,3)   3.0  0.0
[2]  103  104
[3]  2  6  7

 4(4,4)   0.0  1.0
[2]  100  101
```

```
[3]  0  1  5

 5(5,5)   1.0  1.0
[2]  101  102
[4]  1  2  4  6

 6(6,6)   2.0  1.0
[3]  102  103  104
[5]  1  2  3  5  7

 7(7,7)   3.0  1.0
[1]  104
[2]  3  6


ELEMENT
*******
Number of all Elements=5      Array of Elements[1000]

Comments:
--------
ElementLabel,(Index), Number of nodes, Label(index)Nodes generating
Element

 *  Labels Elements over each node
[]- Number of lines generating Element , Labels of this Lines
[]- Number of Elements neighbouring with Element over the Lines ,
Labels of this Elements
--------
  100 (0,0)        nNodes: 3          0(0)        1(1)        4(4)

   *node(0)=  *node(1)= 101 102  *node(4)= 101
  [3]    0/0    1/1    2/2
  [1]    101
............
  101 (1,1)        nNodes: 3          5(5)        4(4)        1(1)

   *node(5)= 102  *node(4)= 100  *node(1)= 100 102
  [3]    3/0    4/1    5/2
  [2]    100    102
............
  102 (2,2)        nNodes: 4   1(1)   2(2)        6(6)        5(5)

 *node(1)= 100 101  *node(2)= 103  *node(6)= 103 104  *node(5)= 101
  [4]    6/0    7/1    8/2    9/3
  [2]    103    101
............
  103 (3,3)        nNodes: 3          2(2)        3(3)        6(6)

   *node(2)= 102  *node(3)= 104  *node(6)= 102 104
  [3]    10/0    11/1    12/2
  [2]    104    102
............
  104 (4,4)        nNodes: 3          7(7)        6(6)        3(3)

   *node(7)=  *node(6)= 102 103  *node(3)= 103
  [3]    13/0    14/1    15/2
  [1]    103
```

```
LINES
*****
Number of allLines = 16      Array of Lines[1000]

Comments:
--------
Line(Glob,Loc),Label(index)Nodes generating line,LabelElement
generating line, Neighbour Line
--------
  (0,0)      nNodes: 2  0(0) 1(1)           [100]     No neighbour!
  (1,1)      nNodes: 2  1(1) 4(4)           [100]     (4/1)
  (2,2)      nNodes: 2  4(4) 0(0)           [100]     No neighbour!
  (3,0)      nNodes: 2  5(5) 4(4)           [101]     No neighbour!
  (4,1)      nNodes: 2  4(4) 1(1)           [101]     (1/1)
  (5,2)      nNodes: 2  1(1) 5(5)           [101]     (9/3)
  (6,0)      nNodes: 2  1(1) 2(2)           [102]     No neighbour!
  (7,1)      nNodes: 2  2(2) 6(6)           [102]     (12/2)
  (8,2)      nNodes: 2  6(6) 5(5)           [102]     No neighbour!
  (9,3)      nNodes: 2  5(5) 1(1)           [102]     (5/2)
  (10,0)     nNodes: 2  2(2) 3(3)           [103]     No neighbour!
  (11,1)     nNodes: 2  3(3) 6(6)           [103]     (14/1)
  (12,2)     nNodes: 2  6(6) 2(2)           [103]     (7/1)
  (13,0)     nNodes: 2  7(7) 6(6)           [104]     No neighbour!
  (14,1)     nNodes: 2  6(6) 3(3)           [104]     (11/1)
  (15,2)     nNodes: 2  3(3) 7(7)           [104]     No neighbour!

MeshTest2.main() - End
```