ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta stavební
Katedra mechaniky

Wangovo dláždění v numerické analýze kompozitů

Wang tiling in numerical analysis of composites

Diplomová práce

Studijní program: Stavební inženýrství
Studijní obor: Konstrukce pozemních staveb

Vedoucí práce: Ing. Anna Kučerová, Ph.D.

Bc. Lukáš Zrůbek

Praha 2012

Zde je prostor pro zadání.

Čestné prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pouze za odborného vedení vedoucí mé diplomové práce Ing. Anny Kučerové, Ph.D.

Dále prohlašuji, že veškeré podklady, ze kterých jsem čerpal, jsou uvedeny v seznamu použité literatury.

Datum: Podpis:

Rád bych velice poděkoval Ing. Anně Kučerové, Ph.D. za ochotu, podporu a trpělivost a za její cenné připomínky a rady při vedení této diplomové práce.

Také moc děkuji Ing. Janu Novákovi, Ph.D. a doc. Ing. Vítu Šmilauereovi, Ph.D. za konzultace a cenné rady.

Rád bych poděkoval i rodině a přítelkyni za podporu, trpělivost a pochopení.

Tato práce byla podpořena projekty číslo P105/11/P370 a P105/10/1682, udělenými Grantovou agenturou České republiky.

Děkuji

Abstrakt

V dnešní době je pro plné využití potenciálu materiálů nutné detailně pochopit charakteristické fyzikální procesy odehrávající se na úrovni mikrostruktury daného materiálu. Ovšem vyhodnocení mikromechanických polí složitých heterogenních mikrostruktur v měřítku makro domény je výpočetně velmi náročný úkol. Jednou z metod, která umožňuje zjednodušení tohoto úkolu, a o které pojednává tato práce, je metoda Wangova dláždění.

Wangovo dláždění využívá princip čtyř-stranných čtvercových dominových hracích kamenů - dlaždic. Spojování těchto dlaždic je povoleno na základě shodné morfologie hran, ovšem dlaždice není možno nijak rotovat nebo zrcadlit. Skupina dlaždic tvoří množinu, která reprezentuje komprimovanou mikrostrukturu. Mikromechanická pole jsou vypočtena na dlaždicích množiny a následně použita pro rekonstrukci mikromechanických polí. Ale protože mechanické veličiny, jako jsou napětí, deformace nebo posuny, jsou nelokální, vznikají v rekonstruovaných mikromechanických polích nespojitosti. Proto jsme do výpočtu mikromechanických polí zahrnuli i sousední dlaždice a vytvořili tak rozšířené množiny dlaždic.

Každou dlaždici standardní množiny dlaždic rozšíříme o všechny přípustné kombinace zahrnutých sousedních dlaždic. Mikromechanická pole vyhodnotíme na těchto dlaždicích se zahrnutými sousedními dlaždicemi. Ze získaných mikromechanických polí jsou odříznuty výsledky pro okolní dlaždice a jsou ponechány pouze mikromechanická pole středových dlaždic, tak zvané dlaždice rozšířené Wangovy množiny. Pro rekonstrukci mikromechanických polí částí nebo celých 2D rovin, jsou využity právě tyto výsledky. Takto zrekonstruovaná mikromechanická pole vykazují menší nespojitosti a průměrné chyby než pole zrekonstruovaná z výsledků spočtených na dlaždicích standardní množiny.

Abstract

Nowadays, to fully exploit the potential of materials, it is necessary to understand in detail to characteristic physical processes taking place at the level of the material microstructure. However, the evaluation of micromechanical fields of complex heterogeneous microstructures in macro-scale domain is a tremendous task. One of the methods, which allows simplification of this task is the method of Wang tiling.

Wang tiling uses a concept of four-sided square dominoes – tiles. The connection of adjacent Wang tiles is allowed through a morphology assigned to congruent edges, but the tiles are not allowed to rotate. Tiles are gathered in set, which represent the compressed microstructure. Micromechanical fields are evaluated on each tile of set and then used to reconstruct micromechanical field of original domain. But because of non-local character of mechanical quantities as stresses, strains or displacements, the reconstructed micromechanical fields contain discontinuities. That is why we add nearest adjacent tiles to evaluating micromechanical fields and created so called extended Wang tiles.

We extend each tile from standard set of tiles sets by every admissible combinations of included adjacent tiles and evaluate micromechanical fields on this combinations. Acquired micromechanical fields are cut off by eight extending tiles and only the results for center tiles of extended tiles are stored. These are called tiles of extended Wang tile set and can be further used for reconstruction of micromechanical field of original domain, which contains less discontinuities and the error is smaller than in micromechanical fields reconstructed from results obtained on tiles from standard set.

Klíčová slova

Wangovo dláždění, dlaždice, mikrostruktury, rozšířené množiny Wangových dlaždic, Fastest Fourier Transform in the West, programovací jazyk C/C++

Keywords

Wang tilings, tiles, microstructures, extended sets of Wang tiles, Fastest Fourier Transform in the West, programming language C/C++

Obsah

Seznam tabulek

Seznam obrázků

Kapitola 1

Úvod

V dnešní stále se zrychlující době je kladen velký důraz na vývoj nových technologií, rozvoj technologií stávajících a výzkum. Také jsou kladeny stále větší nároky a požadavky na koncové produkty. Trend stupňujících se požadavků je nejvíce způsoben trhem a konkurenčním prostředím, kde je nutné pro uplatnění se, nabízet nové, lepší a levnější produkty či služby. Bohužel požadavky mohou být, a velmi často i jsou, protichůdné a ne vždy je možné nabídnout produkt kvalitnější a s nižšími náklady. Nicméně pro alespoň částečné splnění těchto požadavků je nutné získávat lepší znalosti o chování materiálů a využít tak jejich vlastnosti na maximum. Jako konkrétní příklad z oboru stavebnictví je možné uvést materiál beton. Základními složkami betonu jsou voda, plnivo, pojivo a další dnes již nenahraditelné složky pro zlepšení konkrétních vlastností, jak čerstvé betonové směsi tak zatvrdlého betonu. Z počtu vyjmenovaných složek lze odvodit, že beton je materiálem heterogenním. Ovšem pokud pracujeme s betonem při statických výpočtech standardních "makro"konstrukcí, považujeme ho za homogenní materiál s jednotnými fyzikálními i mechanickými vlastnostmi v každém bodě. Tento předpoklad sebou přináší výhody jako je především zjednodušení výpočtů, ale i určité nevýhody ve formě vnášení nepřesností do výpočtu. Tyto nepřesnosti však hrají rozhodující úlohu při modelování komplexních konstrukcí, jejichž kolaps představuje nadstandardní riziko obecného ohrožení, obrovských finančních nebo kulturních ztrát. Jedná se zejména o modelování vodních nádrží, tunelů, jaderných elektráren, mostů a významných historických budov. Lze tedy namítnout proč v dnešní době počítačů nepoužíváme k výpočtům přesné makro nebo mikro struktury a přesné vlastnosti jednotlivých složek. Bohužel i s velkým výpočetním výkonem dnešních počítačů by byly takové výpočty časově velmi náročné a především neefektivní. Jak tedy získat potřebné znalosti o chování betonu, o probíhajících dějích a o mechanických vlastnostech na mikroskopické úrovni? Jednou z metod, kterou lze zjednodušit náročné výpočty, a o které pojednává i tato práce, je metoda takzvaného Wangova dláždění popsaná v kapitole 3, kde je vysvětlen princip, způsob tvorby dlaždic, jednotlivé pojmy a použití základních a rozšířených množin dlaždic při výpočtech. V kapitole 4 je popsán princip výpočtů

mechanických polí s využitím homogenizace, Wangova dláždění, Fourierovy transformace a volně šiřitelné knihovny funkcí pro výpočty Fourierovy transformace s názvem FFTW. V kapitole 5 je popsán mnou vytvořený program, popis proměnných, algoritmu a důležitých funkcí a částí použitých pro výpočty. Kapitola 6 zobrazuje získaná data z výpočtů, především spočtená mikromechanická pole a průměrné chyby pro použité množiny dlaždic.

Kapitola 2

Modelování heterogenních materiálů

Existuje celá řada metod, které se zabývají modelováním mechanické odezvy v heterogenních materiálech. Podrobněji jsou používané metody rozepsány v [5]. My se v této kapitole zaměříme na numerickou homogenizaci, která je pravděpodobně nejpoužívanější metodou v současné době, opět podrobnější vysvětlení a členění samotného homogenizačního přístupu lze nalézt v [7] a základy této matematické metody lze nalézt mimo jiné v [1].

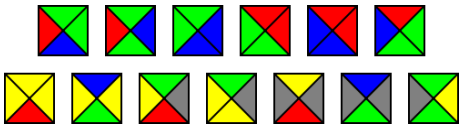
Numerická homogenizace je účinným nástrojem pro odvození efektivních modelů v měřítku našeho zájmu. Materiálové heterogenity jsou na tzv. mezo- nebo mikro-úrovni charakterizovány pomocí periodické jednotkové buňky (PUC) [17] nebo pomocí statisticky ekvivalentní periodické jednotkové buňky (SEPUC) [21]. Využití homogenizace však předpokládá významné oddělení měřítek mezi makro- a mikro-úrovní, což není u mnoha stavebních materiálů možné splnit (např. v případě betonu či zdiva). Homogenizace se přesto používá i v těchto případech, ačkoliv tím dochází ke vnášení dalších nepřesností do výpočtu.

Naproti tomu lze využít nové metody, které nepracují na mikro-úrovni pouze s jednou periodickou buňkou, ale reprezentují médium pomocí množiny buněk, tzv. dlaždic. Dlaždice jsou vzájemně kompatibilní a pomocí určitého algoritmu mohou pokrýt nekonečně velkou oblast tak, aby tato oblast nebyla periodická. Tato metoda se nazývá Wangovo dláždění a bude vysvětleno podrobněji v následující kapitole.

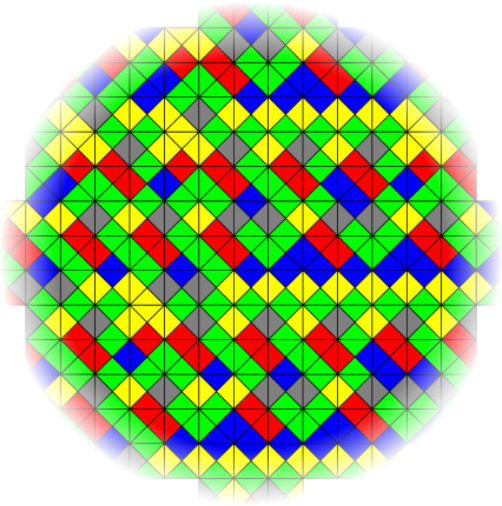
Kapitola 3

Wangovo dláždění

Zmíněnou metodou je v mechanice neznámá metoda Wangových dlaždic. Poprvé tuto myšlenku představil matematik, logik a filozof Hao Wang v roce 1961 [18]. Zjednodušeně lze Wangovo dláždění přirovnat ke hře domino. Hrací kameny (kostky) v tomto případě představují stejně velké čtvercové dlaždice s rozdílnou informací uloženou na hranách, například různé barvy (viz obrázek č. 3.1). Takovéto dlaždice je možné k sobě přikládat na základě shodných informací/barev jednotlivých hran, avšak není dovoleno dlaždice nijak rotovat ani zrcadlit. S určitým počtem vhodně vybraných dlaždic (množinou) je možné vytvořit neomezeně velké rovinné neperiodické mozaiky (viz obrázek č. 3.2).



Obrázek 3.1: Příklad Wangových dlaždic [19].



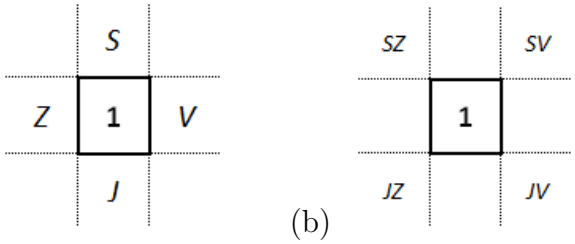
Obrázek 3.2: Příklad mozaiky vytvořené pomocí Wangových dlaždic [19].

3.1 Dlaždice

Jak již bylo popsáno v úvodu této kapitoly, Wangovo dláždění lze přirovnat ke hře domino, kde dva k sobě přiložené kameny musí mít shodný počet značek (shodnou informaci) na polovinách, kterými kameny sousedí. V případě Wangova dláždění nazýváme kameny dlaždice, které jsou čtvercového tvaru a lze je tak k sobě přikládat ve čtyřech různých směrech. Pro snadnější orientaci nazvěme tyto směry podle světových stran sever (S), jih (J), východ (V) a západ (Z) (viz obrázek 3.3a). Každá dlaždice musí tedy obsahovat čtyři informace, jednu v každém směru, podle kterých lze rozhodnout zda je možné umístit dlaždice vedle sebe. Tyto informace nejsou jako v dominu uloženy na celých polovinách dlaždic, ale pouze na jednotlivých hranách. Z toho vyplývá, že dvě přilehlé dlaždice nemusí být totožné, stačí pokud mají shodnou hranovou informaci. Informace může být reprezentována například barvou, řeckým písmenem jako na obrázku 3.4, číslicí, atd. Dlaždice není dovoleno nijak rotovat ani zrcadlit, proto jsou dvě dlaždice se stejnou kombinací hranových informací, z nichž jedna je otočená o násobky $\pi/4$, považovány za různé.

3.2 Wangova množina dlaždic

Wangovy dlaždice se seskupují do takzvaných Wangových množin s konečným počtem dlaždic. Takovou množinu označujeme například W8/2-2. První číslice za písmenem W (Wang) znamená, že se množina skládá z 8 unikátních dlaždic. Číslice za lomítkem označují počet hranových informací n_i^c, kde i reprezentuje svislý nebo vodorovný směr. Konkrétně, v množině s označením W8/2-2 se vyskytují na svislých hranách dlaždic 2 různé hranové informace $\{\alpha, \gamma\}$ a na vodorovných hranách také 2 hranové informace $\{\beta, \delta\}$ (viz obrázek 3.4). Každá z hranových informací je v množině zastoupena stejnou frekvencí výskytů $q_\alpha = q_\beta = q_\gamma = q_\delta = \frac{1}{4}$.



(a) (b)

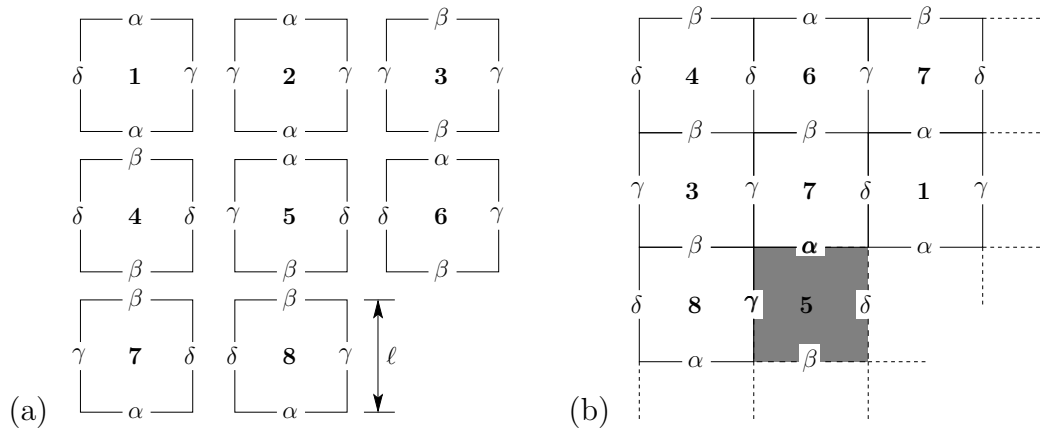
Obrázek 3.3: (a) Základní směry návaznosti dlaždic, (b) vedlejší směry návaznosti dlaždic.

Množiny dlaždic lze rozdělit na minimální, nekompletní a kompletní. Počet dlaždic v kompletní množině n^{cs} je roven všem možným kombinacím dlaždic v závislosti na počtu hranových informací n_i^c (viz rovnice 3.1). V minimální množině je počet dlaždic

n^t (rovnice 3.2) závislý na počtu dlaždic v kompletní množině n^{cs} a počtu možností výběru při umísťování dlaždice do rohu mezi dvě dlaždice již umístěné, například n^{SZ}. Rohové pozice nazýváme podle kombinace světových stran, tedy severo-západ (SZ), severo-východ (SV), jiho-západ (JZ) a jiho-východ (JV) (viz obrázek 3.3b). Množinu, která je složena z většího počtu dlaždic než množina minimální a zároveň je tento počet dlaždic menší než u množiny kompletní, lze nazvat množinou nekompletní.

$$n^{cs} = (n_1^c n_2^c)^2 \tag{3.1}$$

$$n^t = n^{SZ}\sqrt{n^{cs}} \tag{3.2}$$



Obrázek 3.4: (a) Minimální množina Wangových dlaždic W8/2-2 složená z 8 dlaždic, (b) příklad neperiodického dláždění s vyznačenými severo-západními vazbami na hranách [15].

3.3 Dláždění

Dlážděním označujeme mozaiku vytvořenou skládáním dlaždic vedle sebe, která vyplňuje určitou rovinnou oblast. Pokud se nám podařilo vytvořit takové dláždění, že nikde v prostoru domény nechybí dlaždice, nazýváme toto dláždění validním. Abychom libovolným skládáním dlaždic vytvořili vždy neperiodická validní dláždění, potřebujeme takzvanou aperiodickou množinu dlaždic [3]. Tento předpoklad potřeby přísně aperiodické množiny může být eliminován, pokud se při tvorbě dláždění budeme řídit Cohen-Shade-Hiller-Deussen (CSHD) dláždícím algoritmem [2]. Při dodržení pravidel CSHD algoritmu jsme stále schopni vytvořit neperiodické validní dláždění, a to i s jinými než striktně aperiodickými množinami dlaždic.

Postup tvorby dláždění CSHD algoritmem je následující. Při dláždění rovinné domény začínáme s prázdným prostorem, kam umísťujeme první dlaždici. Protože neexistují pro tuto dlaždici žádná omezení, můžeme z množiny náhodně vybrat jednu

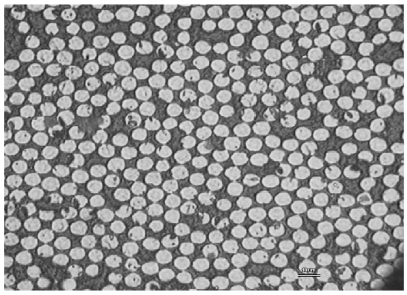
libovolnou. Tato dlaždice nám již určí první podmínky pro umístění dlaždic na další pozice. Pro umístění na jednu z pozic v hlavních směrech (S, J, V, Z), je nutné splnit podmínku shodnosti příslušné hrany. Počet vhodných dlaždic se tedy snížil. Když jako příklad použijeme minimální množinu, je počet vhodných dlaždic na každou z pozic v hlavních směrech roven $n_i^c \cdot 2$. Jednu z vhodných dlaždic náhodně vybereme a umístíme. Umístěním čtyř dlaždic v hlavních směrech získáme čtyři pozice ve vedlejších směrech (SV, SZ, JV, JZ). Pro umístění dlaždice na rohovou pozici je nutné splnit dvě podmínky shodnosti hran. A abychom dodrželi pravidla CSHD algoritmu, je minimální počet dlaždic pro umístění do rohu n^{SZ} roven 2 dlaždicím, aby byla vždy zajištěna možnost výběru. Opět tedy náhodně vybereme jednu z dvojice vhodných dlaždic a umístíme. Opakováním potřebných kroků a dodržováním pravidel CSHD algoritmu vyskládáme požadovanou rovinnou doménu.

n_i^c		n^{cs}	n^{SZ}	n^t	Označení
n_1^c	n_2^c				
2	2	16	2	8	W8/2-2
3	3	81	2	18	W18/3-3
4	4	256	2	32	W32/4-4
5	5	625	2	50	W50/5-5
...	...				

Tabulka 3.1: Počty dlaždic v minimálních Wangových množinách v závislosti na počtu hranových informací (viz rovnice 3.1 a 3.2).

3.4 Standardní množiny Wangových dlaždic

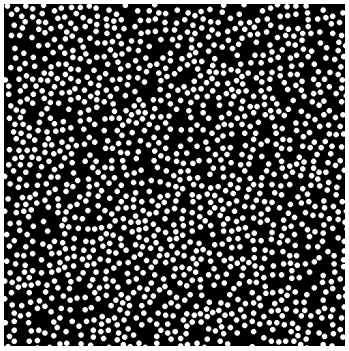
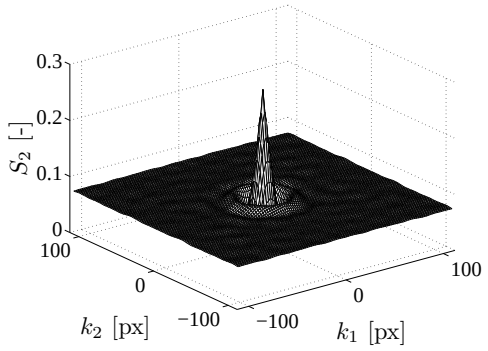
Jelikož v následující podkapitole 3.5 definujeme nový název rozšířené množiny Wangových dlaždic, nazvěme všechny výše popsané množiny Wangových dlaždic jako standardní. Pokud chceme využít Wangova dláždění k výpočtům mikromechanických polí konkrétní mikrostruktury, potřebujeme nejprve vytvořit takovou standardní množinu dlaždic, aby validní dláždění (rekonstruovaná mikrostruktura) vytvořené touto množinou obsahovalo stejné nebo větší množství informací jako původní mikrostruktura.



Obrázek 3.5: Mikrofotografie příčného řezu uhlíkovými vlákny spojených polymerem [20].

3.4.1 Generování dlaždic

Pro zjednodušení vysvětlení, použijme náhodnou mikrostrukturu kompozitu obsahující pouze dvě fáze (viz obrázek 3.6a). Mikrostruktura je složena z homogenní izotropní matrice s náhodně rozdělenými, stejně velkými, kruhovými inkluzemi (disky) o poloměru ρ. Jako materiál odpovídající této mikrostruktuře si lze představit například příčný řez uhlíkovými vlákny spojených polymerem (viz obrázek 3.5). Abychom byli schopni popsat mikrostrukturu jinak než pouze její geometrií a ověřit tak zda námi vytvořené dlaždice dostatečně reprezentují originální mikrostrukturu, použijeme statistický deskriptor nazývaný dvoubodová pravděpodobnost (rovnice 3.5) [15]. Tento deskriptor říká jaká je pravděpodobnost, že dva body o určité vzdálenosti leží ve stejné fázi, v tomto případě ve fázi bílé, reprezentující inkluze. Takto spočítáme pravděpodobnost pro všechny možné vzájemné polohy dvou bodů mikrostruktury a získáme graf zobrazený na obrázku 3.6b).

(a)  (b) 

Obrázek 3.6: (a) Příklad dvoufázové mikrostruktury, (b) graf závislosti dvoubodové pravděpodobnosti S^2 na vzdálenosti bodů.

Při tvorbě množiny dlaždic nejprve navrhneme n^t počátečních dlaždic o hranové délce $\ell \in \mathbb{N}$ v pixelech, ve kterých volně rozložíme n^d disků o poloměru ρ. Každý disk lze prezentovat pomocí vektoru p_i (rovnice 3.3), kde $t_d \in \{1, \ldots, n^t\}$ odpovídá číslu dlaždice a $x_{1,d}, x_{2,d}$ souřadnicím středu disku v dlaždici. Následuje optimalizace těchto dlaždic pomocí metody simulovaného žíhání [10, 16]. Velice zjednodušeně lze říci, že vždy pohneme jedním z n^d disků, pomocí takto upravených dlaždic a CSHD algoritmu zrekonstruujeme originální mikrostrukturu a rovnicí 3.6, vypočítáme chybu geometrie mikrostruktury originální a zrekonstruované.

$$p = [t_d, x_{1,d}, x_{2,d}]_{d=1}^{n^d} \tag{3.3}$$

$$\mathbb{K}^{\mathcal{O}} = \left\{ m \in \mathbb{Z}^2 : -\frac{n_i^{\mathcal{O}}}{2} < m_i \leq \frac{n_i^{\mathcal{O}}}{2}, i \ = 1, 2 \right\}. \tag{3.4}$$

$$S_2(k) = \frac{1}{n_1^{\mathcal{O}} n_2^{\mathcal{O}}} \sum_{m \in \mathbb{K}^{\mathcal{O}}} \chi(m) \chi\left(\lfloor k \ + m \rfloor_{\mathbb{K}^{\mathcal{O}}}\right) \tag{3.5}$$

$$f^S(p) = \frac{1}{n_1^{\mathcal{O}_S} n_2^{\mathcal{O}_S}} \sum_{k \in \mathbb{K}^{\mathcal{O}_S}} \left(S_2(k) - \widetilde{S}_2(p,k) \right)^2 \tag{3.6}$$

Více podrobnějších informací o tvorbě dlaždic, je možné nalézt v článku 'Modelling and Simulation in Materials Science and Engineering' od autorů J. Nováka, A. Kučerové a J. Zemana [15].

3.4.2 Využití

Tímto způsobem získáme dlaždice optimalizované čistě z hlediska geometrie [14]. Pokud na takovýchto dlaždicích spočteme mikromechanická pole, a pokusíme se pomocí nich rekonstruovat celé mikromechanické pole originální mikrostruktury zjistíme, že ač byly dlaždice hranově kompatibilní, spočtená mikromechanická pole jednotlivých dlaždic, již kompatibilní být nemusí. Při rekonstrukci mikromechanického pole originální mikrostruktury vznikají na hranách jednotlivých vedle sebe umístěných mikromechanických polí nespojitosti. V článku [15] je popsán možný postup optimalizovat dlaždice z hlediska geometrie a mikromechanických polí zároveň sloučením obou kritérií váhováním do jedné minimalizované funkce. Bohužel tato dvě kritéria jdou velmi často proti sobě a dochází ke vzniku periodického opakování disků v rekonstruované mikrostruktuře.

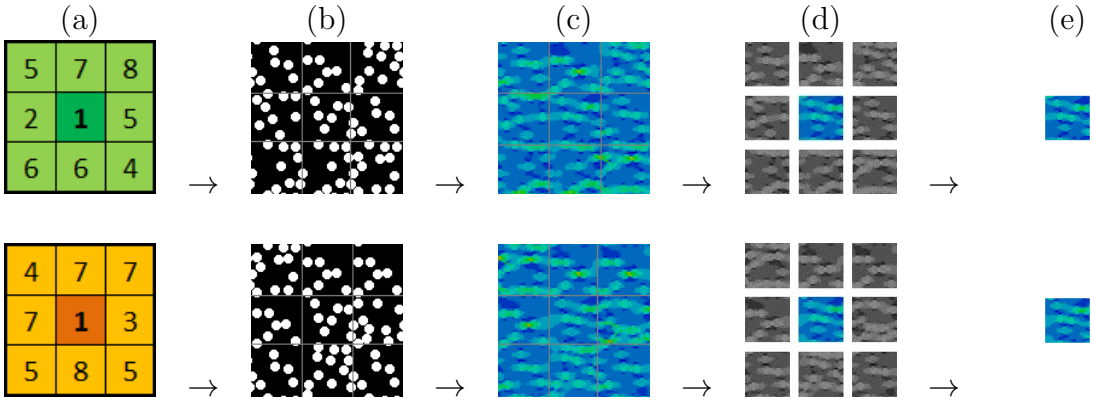
3.5 Rozšířené množiny Wangových dlaždic

Naprosto novým přístupem, kterým by mělo být možné počítat kompatibilní mikromechanická pole, a který by měl být ověřen v této diplomové práci, jsou rozšířené množiny Wangových dlaždic. Principem je použití standardní Wangovy množiny optimalizované pouze z hlediska geometrie a při výpočtu mikromechanických polí jednotlivých dlaždic zahrnout i konečný počet vrstev okolních dlaždic. Pro vysvětlení uvažujme rozšíření o jednu okolní vrstvu na 3×3 dlaždice jak je zobrazeno na obrázku 3.7.

2	**1**	6	3	4	8	3	6	4
2	8	6	3	3	5	7	8	5
6	5	2	7	7	2	**1**	5	8
2	2	2	7	**1**	6	6	4	7
4	8	6	3	4	2	2	7	**1**
3	3	4	7	7	2	8	5	8
1	5	7	**1**	3	4	**1**	6	5
8	6	5	8	5	**1**	4	8	4
1	4	2	7	2	2	7	**1**	5

Obrázek 3.7: Mapa dláždění s vyznačeným rozšířením na 3×3 dlaždice.

Při výpočtu mikromechanického pole pro zelenou nebo oranžovou dlaždici číslo 1, rozšíříme tuto dlaždici o její nejbližší sousední dlaždice (obrázek 3.8a) na hlavních (S, J, V, Z) a vedlejších pozicích (SV, SZ, JV, JZ). Vytvoříme příslušnou mikrostrukturu složenou ze 3×3 dlaždic (obrázek 3.8b) a pro tuto mikrostrukturu vypočteme mikromechanické pole (obrázek 3.8c). Z vypočtených výsledků nás zajímají pouze výsledky odpovídající středové dlaždici, proto můžeme výsledky odpovídající dlaždicím okolním odříznout (obrázek 3.8d). Výsledkem je dlaždice rozšířené Wangovy množiny dlaždic (obrázek 3.8e).



Obrázek 3.8: (a) Dlaždice s rozšířením na 3×3, (b) mikrostruktura 3×3, (c) vypočtená mikromechanická pole, (d) šedě označené výsledky k odříznutí, (e) výsledná dlaždice rozšířené množiny Wangových dlaždic.

Množina	Počet možností výběru dlaždic									Počet kombinací
	střed	S	J	V	Z	SZ	SV	JZ	JV	$n_{3\times3}^{comb}$
W8/2-2	8	4	4	4	4	2	2	2	2	$32,768$
W18/3-3	18	6	6	6	6	2	2	2	2	$373,248$
W32/4-4	32	8	8	8	8	2	2	2	2	$2,097,152$
W50/5-5	50	10	10	10	10	2	2	2	2	$8,000,000$

Tabulka 3.2: Počty možností výběru dlaždic na jednotlivé pozice a počty možných kombinací pro minimální Wangovy množiny při rozšíření na 3×3 dlaždice.

Na výslednou dlaždici rozšířené Wangovy množiny má tedy vliv kombinace dlaždic okolních. Těchto kombinací je velké množství (viz tabulka 3.2), proto existuje i velké množství dlaždic v rozšířené množině. Pro každou kombinaci dlaždic 3×3, která splňuje podmínky validního dláždění, existuje jedna dlaždice rozšířené množiny Wangových dlaždic. Velmi důležité je v tomto případě označení jednotlivých dlaždic, které můžeme provést například pro zelenou dlaždici číslo 1 kódem 1-57854662. Číslice na první pozici představuje číslo středové dlaždice a následujících osm číslic nám dává informaci o tom jaké byly dlaždice sousední v tomto pořadí pozic $SZ - S - SV - V - JV - J - JZ - Z$.

Po získání všech dlaždic rozšířené Wangovy množiny můžeme rekonstruovat mikromechanické pole. Postupujeme podle mapy dláždění (obrázek 3.7), kdy vybíráme vždy takovou dlaždici z rozšířené množiny, aby měla sousední dlaždice odpovídající sousedním dlaždicím v mapě dláždění.

3.5.1 Rozšiřování o více vrstev

Princip rozšířených množin Wangových dlaždic jsme vysvětlili na příkladu rozšíření o jedinou vrstvu okolních dlaždic, tedy na rozměr 3×3. Toto rozšíření však můžeme zvětšit na dvě vrstvy okolních dlaždic, čímž dostaneme rozměr 5×5 (obrázek 3.9), na tři vrstvy a tedy rozměr 7×7, na čtyři vrstvy a tak dále. Samozřejmě se zvětšujícím se počtem vrstev rozšíření se zvětšuje i počet možných kombinací okolních dlaždic a tedy i počet dlaždic v rozšířené Wangově množině $n_{m\times m}^{comb}$, který lze určit podle rovnice 3.7. Přehled počtu kombinací pro zvětšující se rozšíření je zobrazen v tabulce 3.3.

Pro výpočty v této diplomové práci bylo použito pouze rozšíření na rozměr 3×3. Zda dojde ke zlepšení spojitosti mikromechanických polí rozšířením o více než jednu vrstvu nebylo prozatím ověřeno.

Obrázek 3.9: Mapa dláždění s vyznačeným rozšířením na 5×5 dlaždic.

$$n_{m\times m}^{comb} = n^t \cdot (2n_i^c)^{2(m-1)} \cdot (n^{SZ})^{(m-1)^2} \tag{3.7}$$

Množina	Počet kombinací $n_{m\times m}^{comb}$ pro velikost rozšíření $m \times m$					
	1×1	3×3	5×5	7×7	9×9	11×11
W8/2-2	$8.0 \cdot 10^0$	$3.3 \cdot 10^4$	$3.4 \cdot 10^{10}$	$9.2 \cdot 10^{18}$	$6.3 \cdot 10^{29}$	$1.1 \cdot 10^{43}$
W18/3-3	$1.8 \cdot 10^1$	$3.7 \cdot 10^5$	$2.0 \cdot 10^{12}$	$2.7 \cdot 10^{21}$	$9.4 \cdot 10^{32}$	$8.3 \cdot 10^{46}$
W32/4-4	$3.2 \cdot 10^1$	$2.1 \cdot 10^6$	$3.5 \cdot 10^{13}$	$1.5 \cdot 10^{23}$	$1.7 \cdot 10^{35}$	$4.7 \cdot 10^{49}$
W50/5-5	$5.0 \cdot 10^1$	$8.0 \cdot 10^6$	$3.3 \cdot 10^{14}$	$3.4 \cdot 10^{24}$	$9.2 \cdot 10^{36}$	$6.3 \cdot 10^{51}$

Tabulka 3.3: Počty možných kombinací pro minimální Wangovy množiny při zvětšujícím se rozšíření zahrnutých dlaždic.

Kapitola 4

Výpočty

Pro ověření metody rozšířených množin Wangových dlaždic a výsledků získaných touto metodou byl vytvořen program popsaný v kapitole 5 pomocí něhož byly provedeny následující výpočty.

Při výpočtech předpokládáme izotropní, lineárně pružné materiály (fáze). Pro každou fázi je nutné zadat jako jediné vstupní hodnoty Youngův modul pružnosti E a Poissonovo číslo ν. V našem případě pracujeme s binárními mikrostrukturami pouze o dvou fázích, kdy jsme poměr Youngova modulu pružnosti fází zvolili v poměru 10 : 1. Vstupní hodnoty pro jednotlivé fáze jsou zobrazeny v tabulce 4.1.

Fáze		E	ν
i	barva		
0	černá	10	0.4
1	bílá	1	0.1

Tabulka 4.1: Hodnoty Youngova modulu pružnosti a Poissonova čísla pro jednotlivé fáze.

Pro každý pixel (voxel - částice objemu představující hodnotu v pravidelné mřížce $3D$ prostoru) je nutné vypočítat matici tuhosti. Výpočet matice tuhosti D pro každý pixel (voxel) probíhá podle následujícího vzorce:

$$D = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5-\nu & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5-\nu & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5-\nu \end{bmatrix} \tag{4.1}$$

Zatížení mikrostruktury je vyvoláno jednotkovou deformací v požadovaném směru. Jeden zatěžovací stav je tedy představován tenzorem deformací ε ve tvaru dle Voighta

(rovnice 4.2), kde jsou všechny složky kromě jedné rovny 0.0 a složka v jejímž směru požadujeme vyvodit zatížení je rovna 1.0.

$$\varepsilon^i = \{\varepsilon_x, \varepsilon_y, \varepsilon_z, \gamma_{yz}, \gamma_{zx}, \gamma_{xy}\}^T \tag{4.2}$$

Protože byl program v rámci této diplomové práce úmyslně tvořen tak, aby byla většina funkcí připravena pracovat s $3D$ vstupními daty, bylo pro práci s $2D$ daty nutné pevně stanovit jeden z rozměrů na hodnotu 1. Z programovacích důvodů popsaných v části 5.2.3 toto není možné provést pro rozměr z. Proto bylo nutné nastavit na hodnotu 1 rozměr x a při práci s $2D$ tedy pracujeme v prostoru $y-z$. Z tohoto důvodu bylo třeba přizpůsobit i zatěžovací stavy, abychom dosáhli zatížení ve směru y, z a yz, tedy v rovině bitmapy. Jednotlivé zatěžovací stavy jsou zobrazeny v tabulce 4.2.

Zaťěžovací stav	Složky tenzoru deformací					
	ε_x	ε_y	ε_z	γ_{yz}	γ_{zx}	γ_{xy}
1	0.0	1.0	0.0	0.0	0.0	0.0
2	0.0	0.0	1.0	0.0	0.0	0.0
3	0.0	0.0	0.0	1.0	0.0	0.0

Tabulka 4.2: Zatěžovací stavy.

Vynásobením matice tuhosti D a tenzoru deformací ε (rovnice 4.3), získáme tenzor napětí σ ve tvaru dle Voighta (rovnice 4.4).

$$\sigma^i = D\varepsilon^i \tag{4.3}$$

$$\sigma^i = \{\sigma_x, \sigma_y, \sigma_z, \tau_{yz}, \tau_{zx}, \tau_{xy}\}^T \tag{4.4}$$

Následuje iterační algoritmus (rovnice 4.6) prezentovaný v článku [13] s vynecháním testu konvergence, kde E odpovídá zatěžovacímu stavu, x představuje pixel (voxel) z množiny všech pixelů (voxelů) V v reálném prostoru a ξ představuje odpovídající frekvenci ve Fourierově prostoru. $\mathcal{F}$ označuje Fourierovu transformaci a $\mathcal{F}^{-1}$ inverzní Fourierovu transformaci. Jednotlivá pole převedená do Fourierova prostoru $\hat{\sigma}^i$ a $\hat{\varepsilon}^i$ jsou označena stříškou a $\hat{\Gamma}^0$ představuje Greenův operátor určený pro referenční izotropní materiál s Lamého konstantami λ^0 a μ^0 podle rovnice 4.5. Označení δ_{ij} odpovídá Kroneckerova delta.

$$\hat{\Gamma}^0_{ijkh}(\xi) = \frac{1}{4\mu^0|\xi|^2}(\delta_{ki}\xi_h\xi_j + \delta_{hi}\xi_k\xi_j + \delta_{kj}\xi_h\xi_i + \delta_{hj}\xi_k\xi_i) - \frac{\lambda^0+\mu^0}{\mu^0(\lambda^0+2\mu^0)}\frac{\xi_i\xi_j\xi_k\xi_h}{|\xi|^4} \tag{4.5}$$

$$
\begin{aligned}
&\text{Inicializace:} \\
\varepsilon^0(x) &= E, \forall x \in V, \\
\sigma^0(x) &= D(x) : \varepsilon^0(x), \forall x \in V, \\
&\text{Iterace } i+1 \\
&\text{kde } \varepsilon^i \text{ a } \sigma^i \text{ jsou známé:} \\
\hat{\sigma}^i &= \mathcal{F}(\sigma^i), \\
\hat{\varepsilon}^i &= \mathcal{F}(\varepsilon^i), \\
&\text{Kontrola rozdílu napětí } \sigma^i \text{ a } \sigma^{i+1} \\
\hat{\varepsilon}^{i+1}(\xi) &= \hat{\varepsilon}^i - \hat{\Gamma}^0(\xi) : \hat{\sigma}^i(\xi), \forall \xi \neq 0 \text{ a } \hat{\varepsilon}^{i+1}(0) = E, \\
\varepsilon^{i+1} &= \mathcal{F}^{-1}(\hat{\varepsilon}^{i+1}), \\
\sigma^{i+1}(x) &= D(x) : \varepsilon^{i+1}(x), \forall x \in V
\end{aligned}
\tag{4.6}
$$

Kontrola rozdílu napětí σ^i a σ^{i+1} probíhá podle rovnice 4.7, kde ϵ je výsledná chyba a n_x počet všech pixelů (voxelů).

$$
\epsilon = \frac{\sqrt{\sum_{x=0}^{n_x} (\sigma^i(x) - \sigma^{i+1}(x))^2}}{n_x} \tag{4.7}
$$

Iterační algoritmus je u konce pokud je chyba ϵ menší než zadaná tolerance.

Kapitola 5

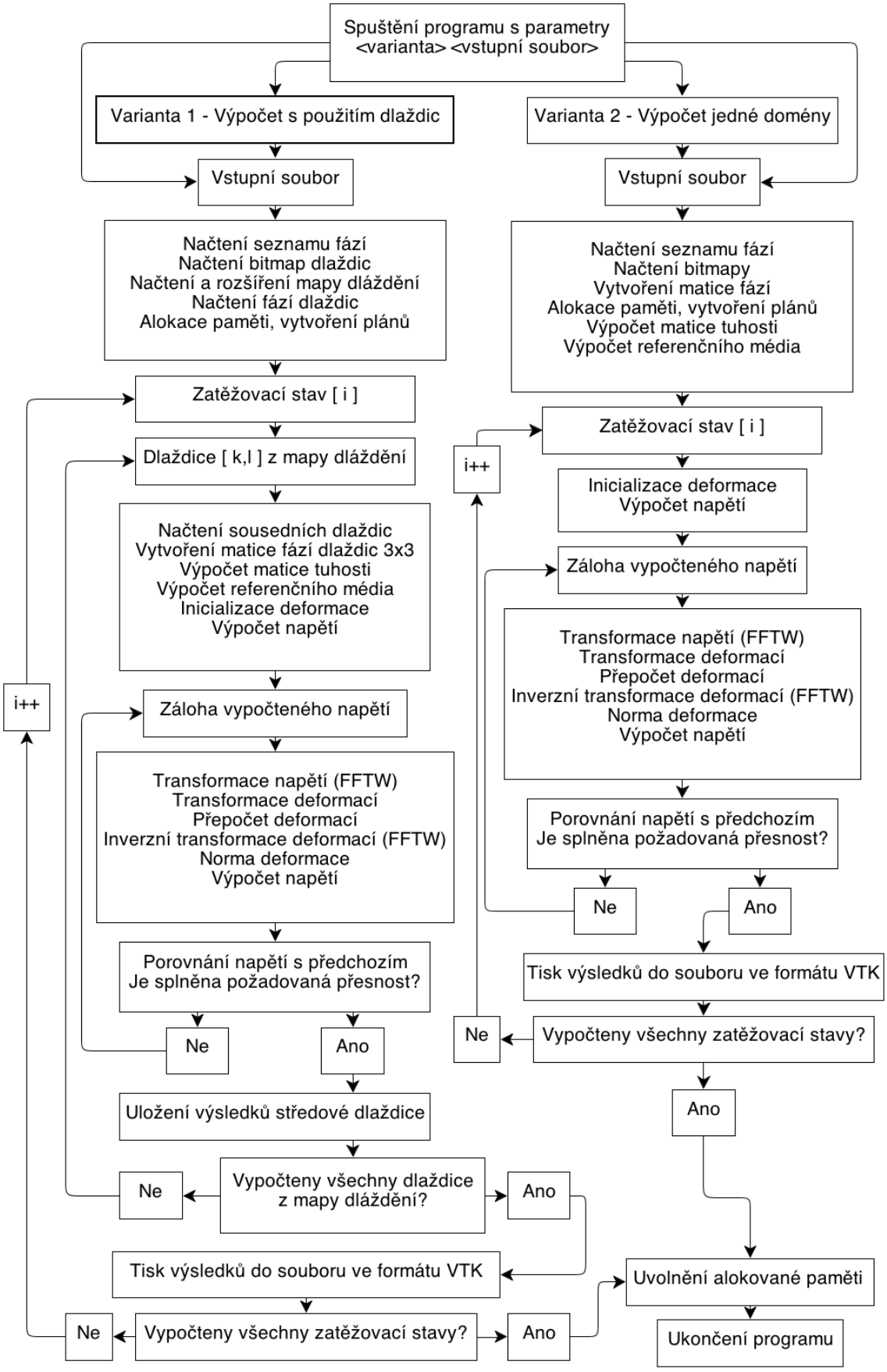
Program pro výpočet

V rámci této diplomové práce byl vytvořen program v programovacím jazyce C/C++ [8, 9, 12] sloužící pro výše popsané výpočty pomocí Wangových dlaždic a rozšířených množin dlaždic. Kód programu vychází z předlohy poskytnuté doc. Ing. V. Šmilauerem, Ph.D., jejímž původním autorem je Mgr. Ing. M. Wierer. Pro aplikaci na náš problém bylo nutné kód výrazně přepracovat a přidat nové potřebné funkce pro práci s dlaždicemi. Program využívá kromě standardních knihoven i několik uživatelských knihoven. Knihovnu pro práci s bitmapami v binárním formátu vytvořenou doc. Ing. J. Zemanem, Ph.D., knihovnu Ing. J. Nováka, Ph.D. pro výstup dat do souboru ve formátu VTK (Visualization Toolkit) [11] a především volně šiřitelnou knihovnu FFTW (Fastest Fourier Transform in the West) [6] pro výpočet rychlé Fourierovy transformace a její inverze, kterou společně vytvořili M. Frigo a S. G. Johnson z MIT (Massachusetts Institute of Technology). Původní předloha docenta Šmilauera pracovala s knihovnou FFTW verze 2.x, která již není dále vyvíjena a byla nahrazena novou verzí FFTW 3.x. Z důvodu ukončení podpory verze 2 byla v rámci tohoto programu využita verze 3, konkrétně FFTW 3.3.3. Bylo ovšem nutné přepracovat funkce využívající tuto knihovnu, protože verze FFTW3 není kompatibilní s FFTW2. Více o knihovně FFTW viz sekce 5.2.3. Zjednodušený princip algoritmu programu je zobrazen na obrázku 5.1.

Program obsahuje dvě varianty výpočtu, varianta 1 slouží k výpočtu pomocí Wangových dlaždic a využívá rozšířených Wangových množin pro rekonstrukci mikromechanického pole. Varianta 2 naopak provede výpočet mikromechanického pole na jediné vstupní bitmapě bez použití Wangových dlaždic.

Ač jsou všechny výpočty a výsledky prezentované v této práci prováděny na bitmapách, tedy v prostoru $2D$, byla snaha program navrhnout tak, aby bylo možné po provedení několika nezbytných úprav kódu provádět výpočty trojrozměrných vstupních dat.

Názvy proměnných a funkcí v programu byly úmyslně voleny v anglickém jazyce, aby netvořily české názvy jazykovou bariéru, pokud by v budoucnosti došlo k uveřejnění programu pro volné užití.



Obrázek 5.1: Diagram průběhu programu.

5.1 Proměnné

Při tvorbě programu byl kladen důraz na umožnění uživateli maximální variabilitu důležitých proměnných. Tyto proměnné je nutno zapsat do vstupního souboru v určeném pořadí a formátu (součástí přiloženého CD jsou i vzorové vstupní soubory pro obě varianty). Uživatel musí tedy do vstupního souboru varianty 1 (Obrázek 5.2) zadat následující proměnné:

- varianta výpočtu ve tvaru `Option_1`
- počet dlaždic množiny ve tvaru celého čísla na samostatném řádku
- názvy/cesty souborů jednotlivých dlaždic, každý soubor na samostatném řádku (binární formát bitmapy)
- název/cesta k souboru obsahující mapu dláždění na samostatném řádku (soubor v textovém formátu)
- počet fází pro načtení ve tvaru celého čísla na samostatném řádku
- název/cesta k souboru obsahující seznam fází, pro každou z fází je nutné zadat číslo fáze, Youngův modul pružnosti (E) a Poissonovo číslo (ν) a to ve tvaru `0 10.0 0.4` (na jednom řádku)
- počet zatěžovacích stavů ve tvaru celého čísla na samostatném řádku
- název/cesta k souboru obsahující seznam zatěžovacích stavů na samostatném řádku, pro každý zatěžovací stav je nutné zadat pořadové číslo a 6 reálných čísel představující požadované zatížení deformací,
  například: `0 1.0 0.0 0.0 0.0 0.0 0.0` (na jednom řádku)
- název výstupního VTK souboru na samostatném řádku (bez české diakritiky, mezer a přípony) např: `Vystup` (za název souboru bude přidáno slovo *load* a číslo zatěžovacího stavu, výsledný název včetně přípony bude tedy vypadat následovně: `Vystup_load1.vtk`)
- požadovanou přesnost výpočtu ve tvaru reálného čísla na samostatném řádku, například: `1.0e-4`
- maximální počet iterací ve tvaru celého čísla na samostatném řádku, například: `1000`
- jméno autora bez české diakritiky na samostatném řádku (bude zobrazeno ve výstupním souboru VTK)
- požadované označení tenzorů napětí ve výstupním souboru VTK

- požadované označení tenzorů deformací ve výstupním souboru VTK

```
Option_1
8
W08-2-2_010_007_tile_1.bmp
W08-2-2_010_007_tile_2.bmp
W08-2-2_010_007_tile_3.bmp
W08-2-2_010_007_tile_4.bmp
W08-2-2_010_007_tile_5.bmp
W08-2-2_010_007_tile_6.bmp
W08-2-2_010_007_tile_7.bmp
W08-2-2_010_007_tile_8.bmp
tiling_map.txt
2
phases.txt
1
load_cases.txt
Vystup_opt1
1.0e-4
1000
Zrubek Lukas
Stress_opt1
Strain_opt1
```

Obrázek 5.2: Příklad vstupního souboru pro variantu 1.

Vstupní soubor pro variantu 2 (Obrázek 5.3) se od varianty 1 liší jen velmi málo. Na prvním řádku vstupního souboru pro variantu 2 musí být text `Option_2`. Dále soubor neobsahuje narozdíl od varianty 1 řádek s počtem dlaždic, řádky se jmény souborů dlaždic a řádek se jménem souboru mapy dláždění. Tyto řádky jsou nahrazeny pouze jedním řádkem s názvem/cestou k souboru jedné vstupní bitmapy (binární formát bitmapy). Následující řádky vstupního souboru se již neliší od varianty 1.

```
Option_2
single_bitmap.bmp
2
phases.txt
1
load_cases.txt
Vystup_opt2
1.0e-4
1000
Zrubek Lukas
Stress_opt2
Strain_opt2
```

Obrázek 5.3: Příklad vstupního souboru pro variantu 2.

Pro každou variantu je tedy určen individuální vstupní soubor. Soubor se seznamem fází a soubor se seznamem zatěžovacích stavů může být společný.

5.2 Funkce programu

Po vytvoření všech potřebných vstupních souborů přistoupíme ke spuštění programu. Program spustíme z příkazové řádky zadáním názvu souboru programu a dvou parametrů definujících variantu výpočtu a název vstupního souboru. Například:

```
program.exe 1 vstupni_soubor.txt
```

S přihlédnutím k možnosti, že si uživatel program přejmenuje, mohou se nultý a druhý argument libovolně lišit od výše uvedeného příkladu. Naopak první argument musí být vždy číslice `1` pro variantu 1 a číslice `2` pro variantu 2. Podle tohoto argumentu program spustí odpovídající výpočet. V případě, že uživatel nezadá správný počet argumentů nebo zvolí jinou variantu než `1` a `2`, program vypíše jednoduchou nápovědu v anglickém jazyce. Jako znamení konce výpočtu bylo přidáno systémové pípnutí po dokončení programu. Hlavní funkce programu `main`, stejně jako celý kód programu je přiložen v příloze A.

5.2.1 Varianta 1

Již při spouštění programu je nutné jako jeden z parametrů zadat číslo varianty 1 nebo 2. Zadáním č. 1 řekneme programu, že požadujeme výpočet s využitím dlaždic a rozšířených Wangových množin. Program zkontroluje zda vstupní soubor zadaný druhým parametrem je určen pro variantu 1 a pokud ano začne načítat potřebná data jako je počet dlaždic, názvy souborů dlaždic, počet fází a tak dále. Po načtení všech dat se spustí hlavní funkce pro výpočet varianty 1 `Option_1();`. Tato funkce nejprve načte jednotlivé dlaždice a podle první dlaždice určí jejich rozměr (z definice Wangova dláždění je zaručeno, že dlaždice budou čtvercové a všechny stejně velké). Dále načte fáze jednotlivých dlaždic, vytvoří všechny potřebné proměnné a pole včetně plánů knihovny FFTW (bližší popis viz sekce 5.2.3). Následuje hlavní cyklus přes všechny zatěžovací stavy, který obsahuje cyklus přes všechny dlaždice z mapy dláždění. Každá dlaždice je rozšířena o 8 svých sousedních dlaždic (4 v hlavních směrech a 4 ve vedlejších směrech). Pro tyto 3x3 dlaždice jsou načteny příslušné fáze jednotlivých bodů (pixelů), a pro každý jednotlivý bod vypočtena matice tuhosti D (rovnice 4.1). Pomocí matic tuhosti D a tenzorů deformací ε příslušného zatěžovacího stavu, se podle rovnice 4.3 vypočte tenzor napětí σ, pro každý bod. Algoritmus pokračuje přeuložením vypočteného napětí do náhradního pole, pro pozdější kontrolu chyby. Nyní je spuštěn iterační algoritmus popsaný rovnicí 4.6, který je ukončen pokud je dosaženo požadované přesnosti. Vypočtené tenzory napětí σ a tenzory přetvoření ε jsou uloženy, pouze pro středovou dlaždici. Postup se opakuje pro každou dlaždici z mapy dláždění. Nakonec jsou vypočtené hodnoty zapsány do souboru ve formátu VTK. V případě dvou a více zatěžovacích stavů, pokračuje výpočet pro následující zatěžovací stav. Pokud

jsou vypočteny všechny zatěžovací stavy je výpočet u konce. Tímto způsobem byly získány výsledky pro rekonstruovaná mikromechanická pole prezentovaná v kapitole 6.

5.2.2 Varianta 2

Postup výpočtu varianty 2 je mnohem jednodušší než u varianty 1. Po kontrole vstupního souboru je načtena jediná bitmapa, její fáze, zatěžovací stavy a další vstupní parametry obsažené ve vstupním souboru. Dále pokračuje algoritmus stejným způsobem jako v případě varianty 1 s tím rozdílem, že kód obsahuje pouze cyklus přes všechny zatěžovací stavy. Po dokončení výpočtu všech zatěžovacích stavů je výpočet dokončen. Pomocí této varianty byly získány výsledky pro nerekonstruovaná mikromechanická pole prezentovaná v kapitole 6.

5.2.3 Knihovna FFTW

Jak již bylo několikrát zmíněno, program se neobejde bez knihovny FFTW neboli *Fastest Fourier Transform in the West* což lze volně přeložit jako "Nejrychlejší Fourierova transformace na západě". Knihovna je dílem autorů M. Friga a S. G. Johnsona z MIT, kteří ji poprvé představili v březnu roku 1997 jako verzi 1.0. Po několika dílčích verzích byla v září roku 1998 vydána verze 2.0 a v dubnu roku 2003 verze 3.0, která je stále rozvíjena až do dnešní aktuální verze 3.3.3 používané tímto programem. Knihovna FFTW slouží k výpočtu diskrétní Fourierovy transformace (DFT) a její inverze pomocí efektivního algoritmu rychlé Fourierovy transformace (FFT) [4] za využití znalostí hardwaru počítače, na kterém je transformace spuštěna. FFTW nepoužívá pevný algoritmus, ale přizpůsobí algoritmus aktuálnímu hardwaru tak, aby bylo dosaženo co největšího výkonu. Tento proces probíhá ve dvou krocích. Prvním krokem je vytvoření takzvaného plánu, který zjistí jak nejrychleji spočítat transformaci na aktuálním počítači. Druhým krokem je spuštění plánu pro příslušná vstupní data. Počet spuštění plánu není nijak omezen. FFTW podporuje transformaci dat o libovolné délce, dimenzi a násobnosti. Funkce pro vytváření plánu jsou rozděleny na 3 vstupní prostředí: Basic (základní), Advanced (pokročilé) a Guru (velmi pokročilé). Dále se jednotlivé funkce rozdělují podle požadované dimenze ($1D, 2D, 3D, nD$) a typu transformace. Typ transformace záleží zda jsou vstupní data komplexní (c) nebo reálná (r) a v jakém formátu požadujeme data výstupní.

V tomto programu je využito pokročilého vstupního prostředí a transformace reálných dat na komplexní ($r2c$) a zpět ($c2r$). Funkce jsou zobrazeny na obrázku 5.4. Proměnná `rank` představuje počet dimenzí vstupního pole, v našem případě 3. Ukazatel `*n` odkazuje na pole o počtu prvků `rank` a obsahuje rozměry jednotlivých dimenzí, v našem případě `{X,Y,Z}`. `Howmany` je počet transformací, které chceme provést. Protože jsou naše data složena vždy ze 6 složek (tensor napětí či přetvoření ve formátu

dle Voighta), zadáváme `howmany = 6`. Ukazatele `*in` a `*out` jsou pole vstupních respektive výstupních dat. Pokud každý ukazatel ukazuje na jiné datové pole v paměti, jedná se o tzv. *out-of-place* transformaci (mimo místo). Pokud se ukazatele odkazují na stejné pole jedná se o *in-place* transformaci (na místě), které je využito i v tomto programu. *In-place* transformace sebou přináší výhodu v úspoře alokované paměti, avšak ne zcela poloviční. Pro transformaci na místě je nutné alokovat takové pole, které by bylo schopné pojmout data reálná nebo zhruba polovinu dat komplexních (při transformaci je využito symetrie komplexních dat, proto stačí uložit pouze přibližnou polovinu). Pro $1D$ *in-place* transformaci je nutné, aby pole o n prvcích bylo rozměru `2*(n/2+1)` (dělení se zaokrouhluje směrem dolů, na celé číslo), pro nD transformaci je rozšířena pouze poslední dimenze. V našem případě $3D$ transformace je nutné alokovat pole o velikosti `X*Y*(2*(Z/2+1))*6`. Ukazatele `*inembed` a `*onembed` odkazují na pole o velikosti `rank`, které obsahuje počet prvků v každé dimenzi, které chceme transformovat. V našem případě chceme provést transformaci všech prvků, proto zadáváme hodnotu `NULL`. Proměnné `istride`, `idist`, `ostride` a `odist` říkají funkci, jak jsou data v paměti uspořádána. `Stride` udává vzdálenost prvků jedné transformace v paměti (6`stride = 6`) a `dist` vzdálenost prvního prvku prvního pole a prvního prvku pole následujícího (`dist = 1`).

```
1  fftw_plan fftw_plan_many_dft_r2c( int rank, const int *n, int howmany,
2                                    double *in, const int *inembed,
3                                    int istride, int idist,
4                                    fftw_complex *out, const int *onembed,
5                                    int ostride, int odist,
6                                    unsigned flags );
7  fftw_plan fftw_plan_many_dft_c2r( int rank, const int *n, int howmany,
8                                    fftw_complex *in, const int *inembed,
9                                    int istride, int idist,
10                                   double *out, const int *onembed,
11                                   int ostride, int odist,
12                                   unsigned flags );
```

Obrázek 5.4: Použité funkce pokročilého prostřední knihovny FFTW.

Knihovna FFTW poskytuje i funkce pro alokaci polí v potřebném formátu (viz obrázek 5.5), která jsou ještě dále přizpůsobena pro potřeby hardwaru počítače. Uživatel má možnost zvolit tři typy přesnosti základní proměnné: `float`, `double` a `long double`. V programu jsem použil typ `double`. Dále je nutné rozlišovat komplexní čísla, pro která je připraven uživatelský formát `fftw_complex` o velikosti dvou proměnných `double` a reálná čísla, pro která slouží standardní typ proměnné `double`. Po dokončení výpočtu je třeba alokovanou paměť dealokovat, k čemuž slouží funkce zobrazené na obrázku (viz obrázek 5.6).

```
1 double *fftw_alloc_real(size_t n);
2 fftw_complex *fftw_alloc_complex(size_t n);
```

Obrázek 5.5: Funkce knihovny FFTW pro alokaci paměti.

```
1 void fftw_free(void *p);
2 void fftw_destroy_plan(fftw_plan plan);
```

Obrázek 5.6: Funkce knihovny FFTW pro dealokaci paměti.

Jelikož není knihovna FFTW primárně určena pro použití v operačním systému Windows, je nutné si pro použití knihovny stáhnout z webových stránek `http://fftw.org/` předpřipravenou dynamickou knihovnu a hlavičkový soubor, pro příslušnou přesnost základní proměnné. Program se zkompiluje s hlavičkovým souborem (je nutné věnovat pozornost typu překladače) a při spouštění programu musí být dynamická knihovna ve stejné složce jako program.

5.3 Databáze

Při rekonstrukci opravdu velké rovinné domény, která obsahuje všechny možné kombinace dlaždic $n \times n$, pomocí metody rozšířených množin Wangových dlaždic, je výhodné uspořit čas tím, že použijeme již vyhodnocená mikromechanická pole středové dlaždice. Tato mikromechanická pole je tedy nutno uschovat pro pozdější použití, tzn. vytvořit jejich databázi. Každé mikromechanické pole je uloženo s číslem středové dlaždice a informací jaká byla kombinace dlaždic sousedních, aby bylo možné jej následně správně použít. Jelikož by databáze obsahovala velké množství takovýchto mikromechanických polí, bylo by nutné vědět kde přesně jaké mikromechanické pole leží, aby se při rekonstrukci domény neztrácel čas prohledáváním celé databáze. Tohoto by bylo možné dosáhnout pokud bychom z čísla středové dlaždice a číslic dlaždic sousedních vytvořili jedinečný kód, pomocí kterého by bylo možné snadno najít požadované pole v databázi. Nicméně v mnoha případech rekonstrukce rovinné domény nebude zapotřebí všech možných kombinací. Jako v případě výpočtů prezentovaných v této diplomové práci. Z tohoto důvodu nepracuje mnou navržený program s databází mikromechanických polí, ale využívá výhodnější metodu, kdy jsou spočteny jen ty kombinace, které pro rekonstrukci mikromechanického pole potřebujeme.

Dalším faktorem ovlivňujícím použití databáze může být velikost potřebné paměti. Jak názorně ukazuje tabulka 3.8 je i pro nejmenší množinu dlaždic W8/2 − 2 počet možných kombinací uspořádání dlaždic do tvaru 3×3, roven hodnotě přesahující $32,000$

kombinací. Pokud použijeme trochu základní matematiky, vědomostí o velikosti jednotlivých proměnných programovacích jazyků C/C++ a předpokladu dlaždice o hraně 62 pixelů (odpovídá dlaždicím z množiny dlaždic na obrázku 6.1(i)), získáme přibližnou představu o potřebné velikosti paměti RAM pro uložení výsledků mikromechanických polí středových dlaždic se všemi možnými kombinacemi dlaždic okolních (kompletní rozšířená Wangova množina). Na většině počítačů je velikost proměnné `double` rovna `8B`. Pokud pro každý pixel potřebujeme uložit 6 hodnot typu `double`, potřebujeme pro každý pixel paměť o velikosti `48B`. Při rozměru dlaždice 62x62 pixelů a počtu $32,000$ kombinací je výsledná potřeba alokované paměti RAM rovna přibližně `5.6GB`.

5.4 Budoucnost a další vhodné úpravy

Jsem si vědom, že i přes veškerou snahu může mnou vytvořený program obsahovat chyby jak logické, tak programové. Zkušenější uživatel jazyka C/C++ by jistě v mnoha případech navrhl lepší a efektivnější řešení, ale s jazykem C/C++ jsem se poprvé setkal před zhruba rokem a půl, proto prosím omluvte mé případné nedostatky.

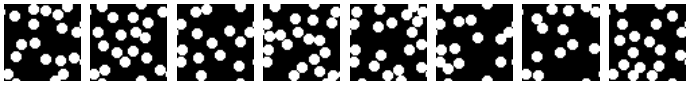
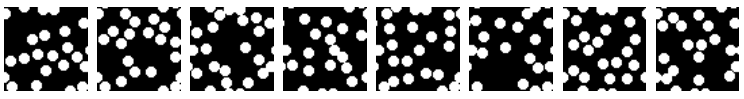
Pokud bude mnou vytvořený program pro někoho přínosem a bude poptávka po jeho rozšíření, mám několik dalších nápadů jak program vhodně upravit, ale bohužel nebyl prostor je realizovat. Pokud bych měl uvést konkrétní příklady, jednalo by se zejména o použití knihovny pro práci s bitmapami s 8bitovou barevnou hloubkou (stupně šedi), čímž by bylo umožněno pracovat s 256 různými fázemi. Další možností je načítání vstupních dat z textového souboru, čímž by bylo umožněno použít libovolný počet fází. Vhodnou úpravou bylo jistě také uzavření celého programu do třídy.

Kapitola 6

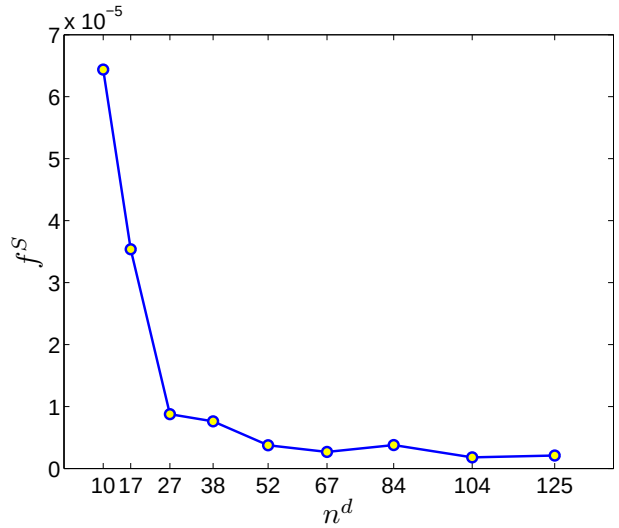
Výsledky

Pro účely ověření metody rozšířených Wangových množin dlaždic byly, za použití výpočtů prezentovaných v kapitole 4 a vytvořeného programu popsaného v kapitole 5, získány výsledky shrnuté v této kapitole.

Abychom mohli vůbec provést potřebné výpočty, bylo nutné zoptimalizovat několik standardních množin Wangových dlaždic. Jedná se o minimální množiny s označením W8/2-2-n^d, tedy o počtu $n^t = 8$ dlaždic a hranových informací $n_i^c = 2$ dlaždic ve svislém nebo vodorovném směru. To, čím se od sebe jednotlivé množiny liší, je počet disků n^d, které obsahují. Se zvyšujícím se počtem disků se zvětšují i jednotlivé dlaždice v množině a prodlužuje se výpočet, který v případě větších dlaždic trval až několik týdnů. Bylo zoptimalizováno celkem 9 množin dlaždic (viz obrázek 6.1).

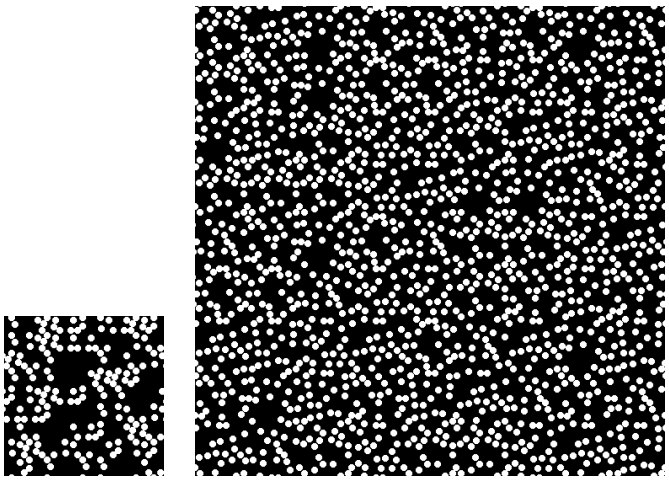
	Množina	Jednotlivé dlaždice množiny
(a)	W8/2-2-010	
(b)	W8/2-2-017	
(c)	W8/2-2-027	
(d)	W8/2-2-038	
(e)	W8/2-2-052	
(f)	W8/2-2-067	
(g)	W8/2-2-084	
(h)	W8/2-2-104	
(i)	W8/2-2-125	

Obrázek 6.1: Označení a jednotlivé dlaždice všech množin použitých k výpočtům.



Obrázek 6.2: Graf závislosti chyby na geometrii f^S (rovnice 3.6) na počtu disků n^d obsažených v množině.

Chyba na geometrii jednotlivých dlaždic (počítaná podle vzorce 3.6 je zobrazena na obrázku 6.2). Jak je možné z grafu vidět, se zvětšujícím se počtem disků n^d obsažených v množině se snižuje velikost chyby na geometrii. Ovšem pro množiny obsahující 84 a 125 disků je chyba větší než pro množinu s nejbližším nižším počtem disků. Tento jev může být způsoben vlivem stochastické optimalizace nebo jsme se již dostali do bodu, kdy začne velikost chyby na geometrii nepatrně oscilovat kolem hodnoty $2.5e^{-6}$.



Obrázek 6.3: (a) Zrekonstruovaná mikrostruktura množinou W8/2-2-010, (b) zrekonstruovaná mikrostruktura množinou W8/2-2-125.

Protože jsou všechny použité množiny vytvořeny tak, aby rozložení hranových informací na jednotlivých dlaždicích bylo stejné pro všechny množiny, můžeme při rekonstrukci mikrostruktur použít jednu validní mapu dláždění (viz tabulka 6.1). Pomocí této mapy a CSHD algoritmu byly zrekonstruovány mikrostruktury na obrázku 6.3.

Všechny vytvořené mikrostruktury jsou zobrazeny v příloze B.

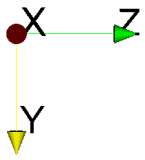
1	0	5	2	3	7	2	5	3
1	7	5	2	2	4	6	7	4
5	4	1	6	6	1	0	4	7
1	1	1	6	0	5	5	3	6
3	7	5	2	3	1	1	6	0
2	2	3	6	6	1	7	4	7
0	4	6	0	2	3	0	5	4
7	5	4	7	4	0	3	7	3
0	3	1	6	1	1	6	0	4

Tabulka 6.1: Validní mapa dláždění použitá pro rekonstrukce mikrostruktur.

Aby bylo možné zrekonstruovaná mikromechanická pole s něčím porovnat a určit tak velikost chyby vzniklou použitím rozšířených množin Wangových dlaždic, byla pomocí 2. varianty vytvořeného programu vypočítána mikromechanická pole vytvořených mikrostruktur (obrázky B.1) vždy jako jeden celek.

Pro názornost jsou v této kapitole prezentovány výsledky množiny s nejmenším počtem disků W8/2-2-010 a množiny s největším počtem disků W8/2-2-125. Všechna vypočtená mikromechanická pole jsou zobrazena v příloze B.

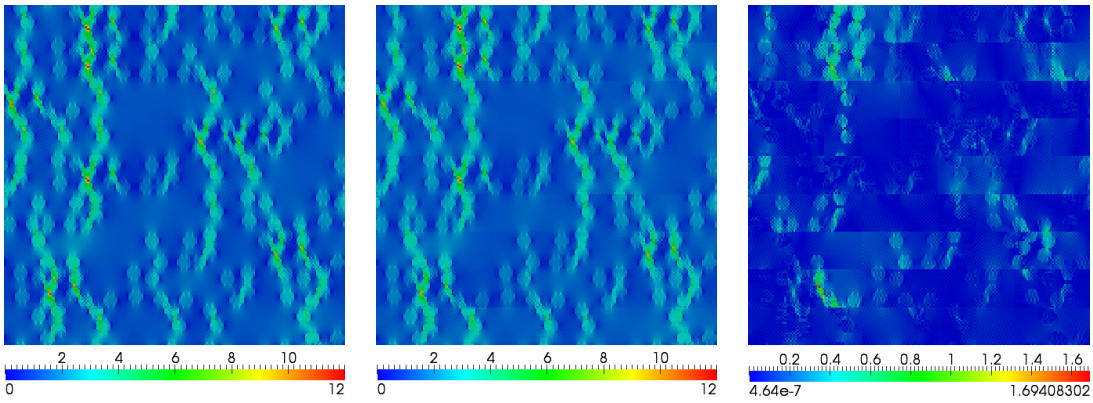
Jak již bylo řečeno v kapitole 4, bylo z programovacích důvodů nutné při $2D$ výpočtech zvolit rozměr x roven 1, a proto se pohybujeme v prostoru $y - z$. Dalším faktem vhodným za zmínku, jsou kladné směry os y a z (viz obrázek 6.4). Zatímco v běžné praxi se ve $2D$ prostoru setkáme s kladnými směry os vycházejících z levého spodního rohu a směřující nahoru a vpravo, v tomto případě vycházejí kladné směry z levého horního rohu a směřují dolů a vpravo.



Obrázek 6.4: Směry.

A nyní již k výsledkům: obrázek 6.5a zobrazuje mikromechanické pole vypočtené na mikrostruktuře jako celku. Konkrétně se jedná o složku napětí σ_y pro mikrostrukturu vytvořenou množinou W8/2-2-010 obsahující nejmenší počet disků ze všech použitých množin, zatíženou jednotkovou deformací ve směru y. Obrázek 6.5b zobrazuje mikromechanické pole zrekonstruované pomocí rozšířené množiny Wangových dlaždic. Opět se jedná o napětí σ_y. Jak si lze na tomto obrázku všimnout, jsou zřetelně vidět nespojitosti jednotlivých dlaždic rozšířené množiny. Ještě lépe jsou nespojitosti mezi dlaždicemi

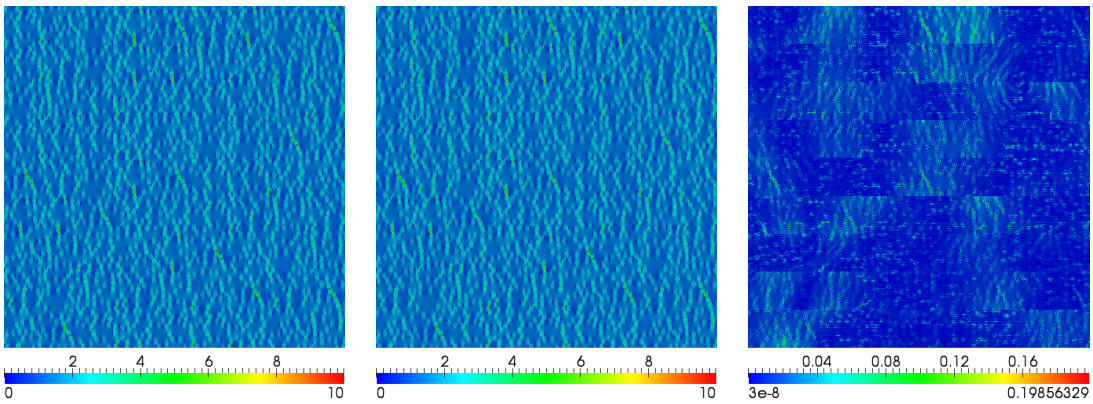
vidět na třetím obrázku 6.5c, který byl získán jako absolutní hodnota rozdílu zrekonstruovaného mikromechanického pole a mikromechanického pole spočteného vcelku (obrázek 6.5a mínus obrázek 6.5b). Vysoká je také hodnota maximální chyby 1.694.



(a) (b) (c)

Obrázek 6.5: Napětí σ_y pro množinu W8/2-2-010 zatížené jednotkovou deformací ve směru y, (a) mikromechanické pole vypočtené v celku, (b) mikromechanické pole zrekonstruované pomocí rozšířené množiny Wangových dlaždic, (c) odečtená mikromechanická pole $a - b$.

Obrázky 6.6a,b zobrazují opět průběh napětí σ_y, ovšem tentokrát pro množinu W8/2-2-125, s největším počtem disků ze všech použitých množin dlaždic. Obrázek 6.6a odpovídá mikromechanickému poli spočtenému jako celek a obrázek 6.6b mikromechanickému poli zrekonstruovanému pomocí rozšířené množiny Wangových dlaždic. Při jejich porovnání pouhým okem se jeví jako identická a ani není možné na obrázku (b) pozorovat nějaké nespojitosti mezi dlaždicemi. Po vytvoření absolutní hodnoty rozdílu obrázku (a) a obrázku (b) je vidět, že se od sebe mikromechanická pole přece jen nepatrně liší, ovšem maximální velikost chyby získaná jako rozdíl dvou sobě odpovídajících pixelů, je pouze 0.198.

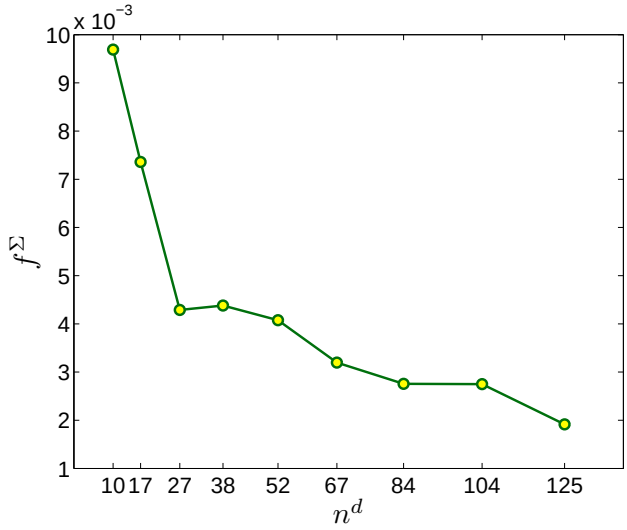


Obrázek 6.6: Napětí σ_y pro množinu W8/2-2-114 zatížené jednotkovou deformací ve směru y, (a) mikromechanické pole vypočtené v celku, (b) mikromechanické pole zrekonstruované pomocí rozšířené množiny Wangových dlaždic, (c) odečtená mikromechanická pole $a - b$.

Lze tedy předpokládat, že se zvětšujícím se počtem disků obsažených v množině, a tedy i zvětšujícími se dlaždicemi množiny, se snižuje chyba zrekonstruovaného mikromechanického pole. Tuto myšlenku je možné ověřit pomocí výpočtu průměrné chyby pro konkrétní množinu, podle rovnic 6.1 a 6.2 z článku [15], kde $\Sigma^*_{ij}(k)$ odpovídá složce napětí ij, k-tého pixelu z množiny všech pixelů $\mathbb{K}^{\mathcal{O}_T}$ mikromechanického pole vypočteného jako celek a $\widetilde{\Sigma}^*_{ij}(k)$ odpovídá složce napětí ij, k-tého pixelu z množiny všech pixelů $\mathbb{K}^{\mathcal{O}_T}$ zrekonstruovaného mikromechanického pole.

$$f^{\Sigma}_{ij}(k) = \frac{\left|\Sigma^*_{ij}(k) - \widetilde{\Sigma}^*_{ij}(k)\right|}{\max\limits_{m\in\mathbb{K}^{\mathcal{O}_T}} \Sigma^*_{ij}(m) - \min\limits_{m\in\mathbb{K}^{\mathcal{O}_T}} \Sigma^*_{ij}(m)}, k \in \mathbb{K}^{\mathcal{O}_T}; i,j \in \{1,2,3\}, \tag{6.1}$$

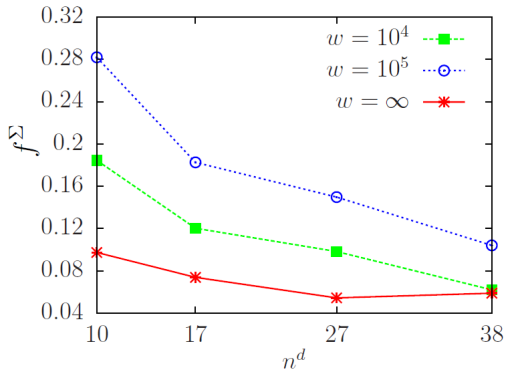
$$f^{\Sigma} = \frac{1}{3}\frac{1}{3}\frac{1}{\mathcal{O}_T}\sum_{i,j=1}^{3}\sum_{k\in\mathbb{K}^{\mathcal{O}_T}} |f^{\Sigma}_{ij}(k)| \tag{6.2}$$



Obrázek 6.7: Graf závislosti průměrné chyby f^{Σ} na počtu disků n^d obsažených v množině.

Vypočtením chyb pro jednotlivé množiny dlaždic, získáme graf (obrázek 6.7) závislosti průměrné chyby f^{Σ} na počtu disků n^d v množině. Z tohoto grafu plyne, že průměrná chyba f^{Σ} se snižuje se zvyšujícím se počtem disků n^d, ovšem stejně jako u grafu (obrázek 6.2) závislosti chyby geometrie na počtu disků n^d, se i zde vyskytují hodnoty neodpovídající předpokládanému uspořádání. Konkrétně velikost průměrné chyby pro množinu obsahující 38 disků (W8/2-2-038) je větší než velikost průměrné chyby pro množinu se 22 disky (W8/2-2/027). Pokud si grafem proložíme spojnici trendu zjistíme, že chyba pro množinu s 22 disky vyšla nadprůměrně menší a z trendu tedy vybočuje tato množina W8/2-2/027 jako průměrně lepší než množiny ostatní. Naopak velikost chyby pro množinu se 104 disky vyšla průměrně větší oproti spojnici trendu. Nicméně postupný trend snižování velikosti průměrné chyby v závislosti na

počtu disků v množině, je z grafu patrný.



Obrázek 6.8: Graf průměrných chyb.

Abychom ověřili, že pomocí metody rozšířených množin Wangových dlaždic lze dosáhnout lepších výsledků než při použití standardních množin Wangových dlaždic, potřebujeme hodnoty pro porovnání. Použijeme proto výsledky prezentované v článku [15] na obrázku 12, kde graf zachycuje velikost průměrné chyby v závislosti na počtu disků v množině, při použití standardních Wangových množin. Získané průměrné chyby (modrá a zelená křivka) zobrazené v grafu (obrázek 6.8), jsou jasně větší než chyby vypočtené pomocí rozšířených množin Wangových dlaždic. Lze tedy říci, že mikromechanická pole vytvořená pomocí rozšířených množin Wangových dlaždic jsou zatížena menší průměrnou chybou než v případě standardních Wangových množin.

Kapitola 7

Závěr

Tato diplomová práce byla od počátku rozdělena na dvě části. Úkolem jedné části bylo ověřit použití rozšířených množin Wangových dlaždic, vliv počtu disků obsažených v množinu na relativní průměrnou chybu rekonstruovaných mikromechanických polí a také porovnání výsledků rozšířených množin Wangových dlaždic s výsledky standardních množin Wangových dlaždic. Úkolem druhé části bylo vytvořit program, pomocí něhož vypočítáme a získáme všechna potřebná data pro první část.

Prvotním cílem bylo spojit a zautomatizovat jednotlivé dílčí kusy kódů vytvořených v průběhu předmětu Projekt 4, za účelem vytvoření potřebného programu. Po získání předlohy programu od doc. Ing. V. Šmilauera, Ph.D. bylo od nápadu slepovat jednotlivé kusy kódů upuštěno a byl stanoven cíl nový, upravit stávající program docenta Šmilauera pro potřeby výpočtů mikromechanických polí. Jak bylo ale zjištěno, program pracoval s již nepodporovanou verzí knihovny FFTW 2. Bylo tedy nutné nastudovat použití knihovny FFTW 3 a upravit stávající kód programu. Následovalo naprogramování všech potřebných funkcí pro práci s dlaždicemi a jejich ověření. Výpočty prováděné programem jsou popsány v kapitole 4. Výsledný program a jeho použití je popsáno v kapitole 5 a v příloze A je zobrazen kód programu samotného.

S využitím hotového programu bylo vypočítáno velké množství výsledků, které byly vyhodnoceny a jsou prezentovány v kapitole 6. Ukázalo se, že se zvyšujícím se počtem disků obsažených v množině, se snižuje i velikost průměrné chyby (viz obrázek 6.7) a tyto chyby jsou menší než pro mikromechanická pole zrekonstruovaná pomocí standardních Wangových množin (viz obrázek 6.8).

V budoucnu by bylo vhodné ověřit zda se bude průměrná chyba i nadále snižovat s rostoucím počtem disků obsažených v množině, nebo zda začne velikost chyby oscilovat. Další zkoumání by si zasloužil i princip rozšiřování zahrnutých dlaždic na více než jednu vrstvu a samozřejmě samotný program by bylo vhodné vylepšit či dále rozšířit.

Literatura

[1] A. Benssousan, J. Lions, and G. Papanicoulau. *Asymptotic Analysis for Periodic Structures*. North-Holland, Amsterdam, 1978.

[2] M. Cohen, J. Shade, S. Hiller, and O. Deussen. Wang tiles for image and texture generation. *ACM Transactions on Graphics*, 22(3):287–294, 2003. ISSN 0730-0301.

[3] K. Culik. An aperiodic set of 13 Wang tiles. *Discrete Mathematics*, 160:245–251, 1996.

[4] P. Duhamel and M. Vetterli. Fast fourier transforms: a tutorial review and a state of the art. *Signal Process.*, 19(4):259–299, Apr. 1990.

[5] W. E and B. Engquist. Multiscale modeling and computation. *Notices of the American Mathematical Society*, 2003. 50(9):1062–1070.

[6] M. Frigo and S. G. Johnson. Fastest fourier transform in the west. `http://fftw.org/`, 2012. [Online].

[7] F. Fritzen. *Microstructural Modeling and Computational Homogenization of the Physically Linear and Nonlinear Constitutive Behavior of Micro-Heterogeneous Materials*. KIT Scientific Publishing, 2012.

[8] P. Herout. *Učebnice jazyka C*. Kopp, 2009.

[9] J. Kent. *C++ bez předchozích znalostí*. Computer Press, 2009.

[10] S. Kirkpatrick, C. J. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983. ISSN 0036-8075.

[11] Kitware. Visualization toolkit. `http://www.vtk.org/`, 2012. [Online].

[12] J. Liberty and B. L. Jones. *Naučte se C++ za 21 dní*. Computer Press, 2007.

[13] H. Moulinec and P. Suquet. A numerical method for computing the overall response of nonlinear composites with complex microstructure. *Computer Methods in Applied Mechanics and Engineering*, 157(1-2):69 – 94, 1998.

[14] J. Novák, A. Kučerová, and J. Zeman. Compressing random microstructures via stochastic wang tilings. *Physical Review E*.

[15] J. Novák, A. Kučerová, and J. Zeman. Microstructural enrichment functions based on stochastic wang tilings. *Accepted for publication - Modelling and Simulation in Materials Science and Engineering*, 2013. e-print: arXiv:1110.4183.

[16] J. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[17] J. Vorel and M. Šejnoha. Evaluation of homogenized thermal conductivities of imperfect carbon-carbon textile composites using the mori-tanaka method. *Structural Engineering and Mechanics*, 33(4):429–446, 2009.

[18] H. Wang. Proving theorems by pattern recognition–II. *Bell Systems Technical Journal*, 40(2):1–41, 1961. ISSN 0005-8580.

[19] Wikipedia. Wang tile. `http://en.wikipedia.org/wiki/Wang_tile`, 2012. [Online].

[20] J. Zeman. Analysis of mechanical properties of fiber-reinforced composites with random microstructure. Master's thesis, Czech Technical University in Prague, 2000.

[21] J. Zeman and M. Šejnoha. From random microstructures to representative volume elements. *Modelling and Simulation in Materials Science and Engineering*, 15(4):S325–S335, 2007. 2007 Higlight paper.

Příloha A

Kód programu v jazyce C/C++

```
/* ==========================================================================

   name:           main.cpp
   description:
   author(s):      Martin Wierer, Jan Zeman, Vit Smilauer, Lukas Zrubek
   place of birth: Czech Technical University in Prague
   last edit:      12/2012
   language:       C, C++

                   This program is distributed in the hope that it will be useful,
                   but WITHOUT ANY WARRANTY
  -------------------------------------------------------------------------- */

/* ==========================================================================
  libraries
  -------------------------------------------------------------------------- */
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <math.h>

  #include "fftw3.h"
  #include "bmp.h"
  #include "postWangTilesVtk.h"
  #include "primaryFunctions.h"

/* ==========================================================================
  global variables
  -------------------------------------------------------------------------- */
  char _INPUT_FILE_[ 100 ]; // name of input file
  char _OUTPUT_FILE_[ 100 ]; // name of output VTK file
  char _TILING_MAP_FILENAME_ [ 100 ]; // name of file containing tiling map
  char _ORIGINAL_[ 100 ]; // name of 'original' file
  char _CREATOR_[ 100 ]; // name of program user
  char Stress_name[ 100 ]; // name of stress tensors in VTK output file
  char Strain_name[ 100 ]; // name of strain tensors in VTK output file
  char _Author_name_[ 100 ]; // name of program user
  char **_TILE_NAMES_; // array of tile file names

  int X = 1; // for 2D computation make this value 1
  int Y; // Y
  int Z; // Z
```

```
43    int _NO_TILES_; // number of tiles in set
44    int tWidth; // width of tile
45    int tHeight; // height of tile
46    int map_rows = 1; // rows of tiling map
47    int map_columns = 1; // columns of tiling map
48    int mtrx[ 3 ][ 3 ]; // 3x3 matrix cut from tiling map
49    int N_loads; // number of load cases
50    int ITERMAX; // iterations limit
51    int iter_num; // number of current iteration
52    int *Microstructure; // array to store phase data (whole reconstructed domain)
53    int **M; // matrix of phases
54    int **tile_map; // tiling map matrix
55    int ***TilePhases; // matrix of phases of each tile
56
57    double ii; //reference medium
58    double scale; // X*Y*Z
59    double E0[ 6 ]; // e11,e22,e33,e23,e13,e12
60    double tolerance; // tolerance
61    double periodx, periody, periodz;
62    double error; // = S_prev - S
63    double *S; // S[ X ][ Y ][ 2*( Z/2+1 )][ 6 ]; = sigma (in place transformation)
64    double *S_prev; // stresses from previous iteration
65    double *E; // E[ X ][ Y ][ 2*( Z/2+1 )][ 6 ]; = epsilon (in place transformation)
66    double **Loads; // matrix of load cases ( [ i ][e11][e22][e33][e23][e13][e12] )
67    double **PhaseMatrix; // list of all used phases [ i ][ 0 ] = E, [ i ][ 1 ] = NY )
68    double **Stress_tensors; // array to store computed data - stress
69    double **Strain_tensors; // array to store computed data - strains
70    double ****D; //stiffness matrix with 21 independent anisotr. elements
71    fftw_plan p_S, p_E, pinv_E; // FFTW plans
72
73  /* ============================================================================
74    function to return position of element in row-major order array
75    i0, i1, i2, i3 = S[ i0 ][ i1 ][ i2 ][ i3 ]
76    ---------------------------------------------------------------------------- */
77  int get_index( int i0, int i1, int i2, int i3 ) {
78    int n1=Y, n2=2*( Z/2+1 ), n3=6; // n0=X not needed
79    return ( i3 + n3*( i2+n2*( i1+n1*i0 )));
80    }
81
82  /* ============================================================================
83    function to return position of element in row-major order array
84    i, j, k = S[ i ][ j ][ k ]
85    ---------------------------------------------------------------------------- */
86  int get_ijk( int i, int j, int k ) {
87    int n1=map_rows*tHeight, n2=map_columns*tWidth; // n0=X not needed
88    return ( k+n2*( j+n1*i ));
89    }
90
91  /* ============================================================================
92    function to extend Tile map ( create periodic layer around )
93    ---------------------------------------------------------------------------- */
94  void extendTileMap() {
95    tile_map[ 0 ][ 0 ] = tile_map[ map_rows ][ map_columns ];
96    tile_map[ 0 ][ map_columns+1 ] = tile_map[ map_rows ][ 1 ];
97    tile_map[ map_rows+1 ][ 0 ] = tile_map[ 1 ][ map_columns ];
98    tile_map[ map_rows+1 ][ map_columns+1 ] = tile_map[ 1 ][ 1 ];
99
100   for( int i=1; i<( map_columns+1 ); i++ ) {
```

```
101     tile_map[ 0 ][ i ] = tile_map[ map_rows ][ i ];
102     tile_map[ map_rows+1 ][ i ] = tile_map[ 1 ][ i ];
103   }
104   for( int i=1; i<( map_rows+1 ); i++ ) {
105     tile_map[ i ][ 0 ] = tile_map[ i ][ map_columns ];
106     tile_map[ i ][ map_columns+1 ] = tile_map[ i ][ 1 ];
107   }
108   }
109
110 /* ============================================================================
111   function to copy 3to3 matrix from Tile map
112   ---------------------------------------------------------------------------- */
113 void cut3to3Matrix( int i, int j ) {
114   mtrx[ 0 ][ 0 ] = tile_map[ i-1 ][ j-1 ];
115   mtrx[ 0 ][ 1 ] = tile_map[ i-1 ][ j ];
116   mtrx[ 0 ][ 2 ] = tile_map[ i-1 ][ j+1 ];
117   mtrx[ 1 ][ 0 ] = tile_map[ i ][ j-1 ];
118   mtrx[ 1 ][ 1 ] = tile_map[ i ][ j ];
119   mtrx[ 1 ][ 2 ] = tile_map[ i ][ j+1 ];
120   mtrx[ 2 ][ 0 ] = tile_map[ i+1 ][ j-1 ];
121   mtrx[ 2 ][ 1 ] = tile_map[ i+1 ][ j ];
122   mtrx[ 2 ][ 2 ] = tile_map[ i+1 ][ j+1 ];
123   }
124
125 /* ============================================================================
126   function to create matrix of phases
127   ---------------------------------------------------------------------------- */
128 void copyPhasestoM() {
129   for( int i=0; i<3; i++ ) {
130     for( int j=0; j<3; j++ ) {
131       for( int k=i*tHeight; k<(i+1)*tHeight; k++ ) {
132         for( int l=j*tWidth; l<(j+1)*tWidth; l++ ) {
133           M[ k ][ l ] = TilePhases[mtrx[ i ][ j ]][ k-(i*tHeight) ][ l-(j*tWidth) ];
134         }
135       }
136     }
137   }
138   }
139
140 /* ============================================================================
141   function to compute stiffness matrix of each pixel/value
142   ---------------------------------------------------------------------------- */
143 void computeStiffnessMatrix() {
144   double e; // Young's modulus
145   double ny; // Poisson's ratio
146   double constant;
147
148   for( int x=0; x<X; x++ ) {
149     for( int y=0; y<Y; y++ ) {
150       for( int z=0; z<Z; z++ ) {
151         e = PhaseMatrix[ M[ y ][ z ] ][ 0 ];
152         ny = PhaseMatrix[ M[ y ][ z ] ][ 1 ];
153         constant = e/(( 1+ny )*( 1-2*ny ));
154
155         D[ x ][ y ][ z ][ 0 ] = D[ x ][ y ][ z ][ 6 ] = \
156         D[ x ][ y ][ z ][ 11 ] = constant*( 1-ny );
157
158         D[ x ][ y ][ z ][ 1 ] = D[ x ][ y ][ z ][ 2 ] = \
```

```
159         D[ x ][ y ][ z ][ 7 ] = constant*ny;
160
161         D[ x ][ y ][ z ][ 15 ] = D[ x ][ y ][ z ][ 18 ] = \
162         D[ x ][ y ][ z ][ 20 ] = constant*(( 1-2*ny )/2);
163
164         D[ x ][ y ][ z ][ 3 ] = D[ x ][ y ][ z ][ 4 ] = D[ x ][ y ][ z ][ 5 ] = \
165         D[ x ][ y ][ z ][ 8 ] = D[ x ][ y ][ z ][ 9 ] = D[ x ][ y ][ z ][ 10 ] = \
166         D[ x ][ y ][ z ][ 12 ] = D[ x ][ y ][ z ][ 13 ] = D[ x ][ y ][ z ][ 14 ] = \
167         D[ x ][ y ][ z ][ 16 ] = D[ x ][ y ][ z ][ 17 ] = D[ x ][ y ][ z ][ 19 ] = 0.0;
168       }
169     }
170   }
171   }
172
173 /* ============================================================================
174   function to compute Sigma=stiffness matrix*epsilon ( S=D*E )
175   ---------------------------------------------------------------------------- */
176 void computeStress() {
177   for( int i=0; i<X; i++ ) {
178     for( int j=0; j<Y; j++ ) {
179       for( int k=0; k<Z; k++ ) {
180       // isotropic case
181         S[ get_index( i, j, k, 0 )] = \
182         D[ i ][ j ][ k ][ 0 ]*E[ get_index( i, j, k, 0 )] + \
183         D[ i ][ j ][ k ][ 1 ]*E[ get_index( i, j, k, 1 )] + \
184         D[ i ][ j ][ k ][ 2 ]*E[ get_index( i, j, k, 2 )];
185
186         S[ get_index( i, j, k, 1 )] = \
187         D[ i ][ j ][ k ][ 1 ]*E[ get_index( i, j, k, 0 )] + \
188         D[ i ][ j ][ k ][ 6 ]*E[ get_index( i, j, k, 1 )] + \
189         D[ i ][ j ][ k ][ 7 ]*E[ get_index( i, j, k, 2 )];
190
191         S[ get_index( i, j, k, 2 )] = \
192         D[ i ][ j ][ k ][ 2 ]*E[ get_index( i, j, k, 0 )] + \
193         D[ i ][ j ][ k ][ 7 ]*E[ get_index( i, j, k, 1 )] + \
194         D[ i ][ j ][ k ][ 11 ]*E[ get_index( i, j, k, 2 )];
195
196         S[ get_index( i, j, k, 3 )] = \
197         2*D[ i ][ j ][ k ][ 15 ]*E[ get_index( i, j, k, 3 )];
198
199         S[ get_index( i, j, k, 4 )] = \
200         2*D[ i ][ j ][ k ][ 18 ]*E[ get_index( i, j, k, 4 )];
201
202         S[ get_index( i, j, k, 5 )] = \
203         2*D[ i ][ j ][ k ][ 20 ]*E[ get_index( i, j, k, 5 )];
204       }
205     }
206   }
207   }
208
209 /* ============================================================================
210   function to initialize strains ( epsilon )
211   ---------------------------------------------------------------------------- */
212 void initializeStrains( int a ) {
213   for( int q=0; q<6; q++ ) {
214     E0[ q ]= Loads[ a ][ q+1 ];
215   }
216
```

```
217   for( int i=0; i<X; i++ ) {
218     for( int j=0; j<Y; j++ ) {
219       for( int k=0; k<Z; k++ ) {
220         for( int q=0; q<6; q++ ) {
221           E[ get_index( i, j, k, q )]=E0[ q ];
222         }
223       }
224     }
225   }
226   }
227
228 /* ============================================================================
229   function to normalize strains ( epsilon )
230   ---------------------------------------------------------------------------- */
231 void normalize() {
232   for( int i=0; i<X; i++ ) {
233     for( int j=0; j<Y; j++ ) {
234       for( int k=0; k<(2*(Z/2+1)); k++ ) {
235         for( int q=0; q<6; q++) {
236           E[ get_index( i, j, k, q )] /= scale;
237         }
238       }
239     }
240   }
241   }
242
243 /* ============================================================================
244   function to store results from previous iteration
245   ---------------------------------------------------------------------------- */
246 void store_prev_results() {
247   for( int i=0; i<X; i++ ) {
248     for( int j=0; j<Y; j++ ) {
249       for( int k=0; k<Z; k++ ) {
250         for( int q=0; q<6; q++ ) {
251           S_prev[ get_index( i, j, k, q ) ] = S[ get_index( i, j, k, q ) ];
252         }
253       }
254     }
255   }
256   }
257
258 /* ============================================================================
259   function to compute error and average stress and strains
260   ---------------------------------------------------------------------------- */
261 int check_stresses( ) {
262   int array_size = X*Y*Z*6;
263   double strain_av[ 6 ] = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 };
264   double stress_av[ 6 ] = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 };
265
266   error = 0.0;
267   for( int i=0; i<X; i++ ) {
268     for( int j=0; j<Y; j++ ) {
269       for( int k=0; k<Z; k++ ) {
270         for( int q=0; q<6; q++ ) {
271           error+=pow( S_prev[ get_index( i,j,k,q )] - S[ get_index( i,j,k,q )], 2 );
272           stress_av[ q ]+=S[ get_index( i, j, k, q )];
273           strain_av[ q ]+=E[ get_index( i, j, k, q )];
274         }
```

```
275       }
276     }
277   }
278
279   for( int i=0; i<6; i++ ) {
280     stress_av[ i ]=stress_av[ i ]/scale;
281     strain_av[ i ]=strain_av[ i ]/scale;
282     if( i>=3 && i<=5 ) {
283       strain_av[ i ]*=2.0; //convert shear eigenstrain
284     }
285     printf( "Stress (%d)=%lf Strain (%d)=%lf\n", i, stress_av[ i ], i, strain_av[ i ]);
286     fflush( stdout );
287   }
288
289   error = sqrt( error ) / array_size;
290   printf( "error = %e tolerance = %e\n", error, tolerance );
291   fflush( stdout );
292
293   if( error<tolerance ) { // when the error is smaller than tolerance, finish
294     return 0; // end of iteration
295   }
296   else {
297     store_prev_results();
298     return 1;
299   }
300   }
301
302 /* ============================================================================
303   function to return gamma operator
304   ---------------------------------------------------------------------------- */
305 double gama_iso( int i, int j, int k, int h, double* x) {
306   double vel;
307   double lambda=0.0;
308   double my;
309   double gama=0.0;
310   my=ii/2; // mu of reference medium
311   vel=x[ 0 ]*x[ 0 ]+x[ 1 ]*x[ 1 ]+x[ 2 ]*x[ 2 ];
312
313   gama = (( 1.0/4.0 ) / ( my*vel )) * ((( k==i ) * x[ h-1 ]*x[ j-1 ]) + \
314   (( h==i ) * x[ k-1 ]*x[ j-1 ]) + (( k==j ) * x[ h-1 ]*x[ i-1 ]) + \
315   (( h==j ) * x[ k-1 ]*x[ i-1 ])) - (( lambda+my ) / ( my*( lambda+2.0*my ))) * \
316   (( x[ i-1 ]*x[ j-1 ]*x[ k-1 ]*x[ h-1 ]) / ( vel*vel ));
317
318   if( my == 0.0 ) {
319     return( 0.0 ); // because in this case gama = NaN
320   }
321   else {
322     return gama;
323   }
324   }
325
326 /* ============================================================================
327   - function computes all eigenvalues and eigenvectors of the matrix
328   - matrix is stored as dense matrix and will be overwritten!
329   - eigenvectors are columns of the matrix evec but matrix is stored
330   in usual way, by rows.
331
332   param a - array containing matrix
```

```
333   param evec - array containing eigenvectors
334   param eval - array containing eigenvalues
335   param n - order of matrix a (number of rows or columns)
336   param ni - maximum number of iterations
337   param ani - number of performed iterations
338   param limit - maximum acceptable absolute value of offdiagonal element
339   ------------------------------------------------------------------------- */
340 void jacobi( double *a, double *evec, double *eval, long n, long ni, long &ani, \
341              double limit ) {
342   long i, j, k, l, ii, jj, en, nn;
343   double c, s, q, r, t, ai, aj;
344
345   en=(n*n-n)/2;
346
347   // initial values
348   k=0;
349   for( i=0; i<n; i++ ) {
350     for( j=0; j<n; j++ ) {
351       evec[ k ]=0.0;
352       k++;
353     }
354   }
355   for( i=0; i<n; i++ ) {
356     evec[ i*n+i ]=1.0;
357   }
358
359   // main iteration loop
360   for( k=0; k<ni; k++ ) {
361     nn=0;
362     for( i=0; i<n; i++ ) {
363       for( j=i+1; j<n; j++ ) {
364         if( fabs( a[ i*n+j ])<limit ) {
365           nn++;
366           continue;
367         }
368         else {
369           q=(a[j*n+j]-a[i*n+i])/2.0/a[i*n+j];
370           r=sqrt(1.0+q*q);
371           if (q>=0.0) {
372             t=-q+r;
373           }
374           else {
375             t=-q-r;
376           }
377           r=sqrt(1.0+t*t);
378           c=1.0/r;
379           s=t*c;
380
381     // A.J
382           ii=i;
383           jj=j;
384           for( l=0; l<n; l++ ) {
385             ai=a[ii];
386             aj=a[jj];
387             a[ii]=ai*c-aj*s;
388             a[jj]=ai*s+aj*c;
389             ii+=n;
390             jj+=n;
```

```
391           }
392
393     // J.A.J
394           ii=i*n;
395           jj=j*n;
396           for( l=0; l<n; l++ ) {
397             ai=a[ii];
398             aj=a[jj];
399             a[ii]=ai*c-aj*s;
400             a[jj]=ai*s+aj*c;
401             ii++;
402             jj++;
403           }
404
405     // Q=Q.J
406           ii=i;
407           jj=j;
408           for( l=0; l<n; l++ ) {
409             ai=evec[ii];
410             aj=evec[jj];
411             evec[ii]=ai*c-aj*s;
412             evec[jj]=ai*s+aj*c;
413             ii+=n;
414             jj+=n;
415           }
416         }
417       }
418     }
419     if( nn==en ) {
420       break;
421     }
422   }
423
424   // number of performed iterations
425   ani=k;
426
427   for( i=0; i<n; i++ ) {
428     eval[ i ]=a[ i*n+i ];
429   }
430   }
431
432 /* ============================================================================
433   function to store data in 2 dimensional array (option 1)
434   ---------------------------------------------------------------------------- */
435 void storeData_opt1( int J, int K ) {
436   for( int i=0; i<X; i++ ) {
437     for( int j=0; j<tWidth; j++ ) {
438       for( int k=0; k<tHeight; k++ ) {
439         for( int q=0; q<6; q++ ) {
440           Stress_tensors[ get_ijk( i, ( J*tWidth )+j ,( K*tHeight )+k )][ q ] = \
441           S[ get_index( i, j+tWidth, k+tHeight, q )];
442
443           Strain_tensors[ get_ijk( i, ( J*tWidth )+j ,( K*tHeight )+k )][ q ] = \
444           E[ get_index( i, j+tWidth, k+tHeight, q )];
445
446           Microstructure[ get_ijk( i, ( J*tWidth )+j ,( K*tHeight )+k )] = \
447           M[ j+tWidth ][ k+tHeight ];
448         }
```

```
449       }
450     }
451   }
452   }
453
454 /* ============================================================================
455   function to store data in 2 dimensional array (option 2)
456   ---------------------------------------------------------------------------- */
457 void storeData_opt2() {
458   for( int i=0; i<X; i++ ) {
459     for( int j=0; j<Y; j++ ) {
460       for( int k=0; k<Z; k++ ) {
461         for( int q=0; q<6; q++ ) {
462           Stress_tensors[ get_ijk( i, j, k )][ q ] = S[ get_index( i, j, k, q )];
463           Strain_tensors[ get_ijk( i, j, k )][ q ] = E[ get_index( i, j, k, q )];
464           Microstructure[ get_ijk( i, j, k )] = M[ j ][ k ];
465         }
466       }
467     }
468   }
469   }
470
471 /* ============================================================================
472   function to print results to VTK
473   ---------------------------------------------------------------------------- */
474 void printVTK( int n ) {
475   FILE *output = NULL;
476   double **points;
477   char VTK_filename[ 100 ];
478   int **cells;
479   int *cell_types;
480   int N = 0, n_points, n_cells;
481
482   sprintf( VTK_filename, "%s_load%d.vtk", _OUTPUT_FILE_, n );
483
484   int X_size = X;
485   int Y_size = map_columns*tWidth;
486   int Z_size = map_rows*tHeight;
487
488   n_points = ( X_size+1 )*( Y_size+1 )*(Z_size+1);
489   n_cells = X_size*Y_size*Z_size;
490
491   points = create_2D_array_double( n_points, 3 );
492   cells = create_2D_array_int( n_cells, 9 );
493   cell_types = new int [ n_cells ];
494
495   N = 0;
496   for( int x=0; x<( X_size+1 ); x++ ) {
497     for( int y=0; y<( Y_size+1 ); y++ ) {
498       for( int z=0; z<( Z_size+1 ); z++ ) {
499         points[ N ][ 0 ] = x/10.0;
500         points[ N ][ 1 ] = y/10.0;
501         points[ N ][ 2 ] = z/10.0;
502         N++;
503       }
504     }
505   }
506
```

```
507   N = 0;
508   int P = 0;
509   for( int x=0; x<X_size ; x++ ) {
510     for( int y=0; y<Y_size; y++ ) {
511       for( int z=0; z<Z_size; z++ ) {
512         cells[ N ][ 0 ] = 8;
513         cells[ N ][ 1 ] = P;
514         cells[ N ][ 2 ] = P+(Z_size+1)*(Y_size+1);
515         cells[ N ][ 3 ] = P+(Z_size+1);
516         cells[ N ][ 4 ] = P+(Z_size+1)*(Y_size+1)+(Z_size+1);
517         cells[ N ][ 5 ] = cells[ N ][ 1 ]+1;
518         cells[ N ][ 6 ] = cells[ N ][ 2 ]+1;
519         cells[ N ][ 7 ] = cells[ N ][ 3 ]+1;
520         cells[ N ][ 8 ] = cells[ N ][ 4 ]+1;
521         N++;
522         P++;
523       }
524       P++;
525     }
526     P++;
527   }
528
529
530   for( int i = 0; i < n_cells; i++ ) {
531     cell_types[ i ] = 11;
532   }
533
534   OpenFile( &output, VTK_filename, "wt" );
535   char fileversion[] = "3.0";
536   char Micro_mark[] = "Microstructure";
537   postWangTilesVtk VTK_output;
538   VTK_output.printVtkHeaderUnstructuredGrid( output, fileversion, _CREATOR_ , \
539                                              Loads[ n ]);
540   VTK_output.printVtkNodalCoordinates( output, points, n_points );
541   VTK_output.printVtkCells( output, cells, n_cells );
542   VTK_output.printVtkCellTypes( output, cell_types, n_cells );
543   VTK_output.printVtk_CELL_DATA_keyword( output, n_cells );
544   VTK_output.printVtkIntScalars( output, Microstructure, Micro_mark, n_cells );
545   VTK_output.printVtkDoubleVoigtMandelTensors( output, Stress_tensors, \
546                                                Stress_name , n_cells );
547   VTK_output.printVtkDoubleVoigtMandelTensors( output, Strain_tensors, \
548                                                Strain_name , n_cells );
549
550   CloseFile( output );
551   printf( "\nVTK file created!\n\n" );
552
553   delete_2D_array( points, n_points );
554   delete_2D_array( cells, n_cells );
555   delete cell_types;
556   }
557
558 // ============================================================================
559 void get_eig_val( double *min, double *max ) {
560   double evec[36];
561   double matrix[36];
562   double matrix1[36];
563   double eval[6];
564   long ani=0;
```

```
565   min[0]=min[1]=10000000;
566   max[0]=max[1]=0;
567
568   //go through all phases in the material
569   for( int i=0; i<X; i++ ) {
570     for( int j=0; j<Y; j++ ) {
571       for( int k=0; k<Z; k++ ) {
572         for(int l=0; l<6; l++ ) {
573           matrix[ l ]=D[ i ][ j ][ k ][ l ];
574         }
575         matrix[ 6 ]=D[ i ][ j ][ k ][ 1 ];
576         for( int l=0; l<5; l++ ) {
577           matrix[ 7+l ]=D[ i ][ j ][ k ][ 6+l ];
578         }
579         matrix[ 12 ]=D[ i ][ j ][ k ][ 2 ];
580         matrix[ 13 ]=D[ i ][ j ][ k ][ 7 ];
581         for( int l=0; l<4; l++ ) {
582           matrix[ 14+l ]=D[ i ][ j ][ k ][ 11+l ];
583         }
584         matrix[ 18 ]=D[ i ][ j ][ k ][ 3 ];
585         matrix[ 19 ]=D[ i ][ j ][ k ][ 8 ];
586         matrix[ 20 ]=D[ i ][ j ][ k ][ 12 ];
587         for (int l=0; l<3; l++ ) {
588           matrix[ 21+l ]=D[ i ][ j ][ k ][ 15+l ];
589         }
590         matrix[24]=D[i][j][k][4];
591         matrix[25]=D[i][j][k][9];
592         matrix[26]=D[i][j][k][13];
593         matrix[27]=D[i][j][k][16];
594         for (int l=0; l<2; l++ ) {
595           matrix[ 28+l ]= D[ i ][ j ][ k ][ 18+l ];
596         }
597         matrix[ 30 ]=D[ i ][ j ][ k ][ 5 ];
598         matrix[ 31 ]=D[ i ][ j ][ k ][ 10 ];
599         matrix[ 32 ]=D[ i ][ j ][ k ][ 14 ];
600         matrix[ 33 ]=D[ i ][ j ][ k ][ 17 ];
601         matrix[ 34 ]=D[ i ][ j ][ k ][ 19 ];
602         matrix[ 35 ]=D[ i ][ j ][ k ][ 20 ];
603
604   //copy matrices
605         for(int l=0; l<36; l++ ) {
606           matrix1[ l ]=matrix[ l ];
607         }
608
609         jacobi( matrix, evec, eval, 6, 200, ani, 0.000001);
610         for( int l=0; l<6; l++ ) {
611           if( eval[ l ]<min[ 0 ]) {
612             min[ 0 ]=eval[ l ];
613             if( eval[ l ]<tolerance ) {
614               printf( "chyba %d %d %d at %d\n", i, j, k, l );
615             }
616           }
617           if( eval[ l ]>max[ 0 ]) {
618             max[ 0 ]=eval[ l ];
619           }
620         }
621   //rotate matrix
622         for( int l=0; l<6; l++ ) {
```

```
623           for( int t=3; t<6; t++ ) {
624             matrix1[ t*6+l ]*=sqrt( 2. );
625             matrix1[ l*6+t ]*=sqrt( 2. );
626           }
627         }
628
629         jacobi( matrix1, evec, eval, 6, 200, ani, 0.000001 );
630         for( int l=0; l<6; l++ ) {
631           if( eval[ l ]<min[ 1 ]) {
632             min[ 1 ]=eval[ l ];
633             if( eval[ l ]<tolerance ) {
634               printf( "chyba %d %d %d at %d\n", i, j, k, l );
635             }
636           }
637           if( eval[ l ]>max[ 1 ]) {
638             max[ 1 ]=eval[ l ];
639           }
640         }
641       }
642     }
643   }
644   }
645
646 // ============================================================================
647 void getFourierStrain( ) {
648   double ksi[ 3 ]={ 0.0, 0.0, 0.0 };
649
650   for( int i=0; i<X; i++ ) {
651     for( int j=0; j<Y; j++ ) {
652       for( int k=0; k<( Z/2+1 ); k++ ) {
653         ksi[ 0 ]=( i-X*(( i>X/2 ) ? 1:0 ))*( 1.0/periodx );
654         ksi[ 1 ]=( j-Y*(( j>Y/2 ) ? 1:0 ))*( 1.0/periody );
655         ksi[ 2 ]=k*( 1.0/periodz );
656
657         if(( i==0 )&&( j==0 )&&( k==0 )) {
658           for( int q=0; q<6; q++ ) {
659             E[ get_index( 0, 0, 0, 2*q )]=E0[ q ]*scale;
660             E[ get_index( 0, 0, 0, 2*q+1 )]=0.0;
661           }
662         }
663         else {
664           for( int q=0; q<3; q++ ) {
665             E[ get_index( i, j, 2*k, 2*q )] -= \
666             (gama_iso( q+1,q+1,1,1,ksi )*S[ get_index( i, j, 2*k, 0 )] + \
667               gama_iso( q+1,q+1,2,2,ksi )*S[ get_index( i, j, 2*k, 2 )] + \
668               gama_iso( q+1,q+1,3,3,ksi )*S[ get_index( i, j, 2*k, 4 )] + \
669               2*(gama_iso( q+1,q+1,1,2,ksi )*S[ get_index( i, j, 2*k, 10 )] + \
670                 gama_iso( q+1,q+1,1,3,ksi )*S[ get_index( i, j, 2*k, 8 )] + \
671                 gama_iso( q+1,q+1,2,3,ksi )*S[ get_index( i, j, 2*k, 6 )]));
672
673             E[ get_index( i, j, 2*k, 2*q+1 )] -= \
674             (gama_iso( q+1,q+1,1,1,ksi )*S[ get_index( i, j, 2*k, 1 )] + \
675               gama_iso( q+1,q+1,2,2,ksi )*S[ get_index( i, j, 2*k, 3 )] + \
676               gama_iso( q+1,q+1,3,3,ksi )*S[ get_index( i, j, 2*k, 5 )] + \
677               2*(gama_iso( q+1,q+1,1,2,ksi )*S[ get_index( i, j, 2*k, 11 )] + \
678                 gama_iso( q+1,q+1,1,3,ksi )*S[ get_index( i, j, 2*k, 9 )] + \
679                 gama_iso( q+1,q+1,2,3,ksi )*S[ get_index( i, j, 2*k, 7 )]));
680           }
```

```
681           for ( int q=1; q<3; q++ ) {
682             for (int l=q+1; l<4; l++) {
683               E[ get_index( i, j, 2*k, 2*(8-q-l) )] -= \
684               (gama_iso(q,l,1,1,ksi)*S[ get_index( i, j, 2*k, 0 )] + \
685                 gama_iso(q,l,2,2,ksi)*S[ get_index(i, j, 2*k, 2 )] + \
686                 gama_iso(q,l,3,3,ksi)*S[ get_index( i, j, 2*k, 4 )] + \
687                 2*(gama_iso(q,l,1,2,ksi)*S[ get_index( i, j, 2*k, 10 )] + \
688                   gama_iso(q,l,1,3,ksi)*S[ get_index( i, j, 2*k, 8 )] + \
689                   gama_iso(q,l,2,3,ksi)*S[ get_index( i, j, 2*k, 6 )]));
690
691               E[ get_index( i, j, 2*k, 2*(8-q-l)+1 )] -= \
692               (gama_iso(q,l,1,1,ksi)*S[ get_index( i, j, 2*k, 1 )] + \
693                 gama_iso(q,l,2,2,ksi)*S[ get_index ( i, j, 2*k, 3 )] + \
694                 gama_iso(q,l,3,3,ksi)*S[ get_index( i, j, 2*k, 5 )] + \
695                 2*(gama_iso(q,l,1,2,ksi)*S[ get_index( i, j, 2*k, 11 )] + \
696                   gama_iso(q,l,1,3,ksi)*S[ get_index( i, j, 2*k, 9 )] + \
697                   gama_iso (q,l,2,3,ksi)*S[ get_index( i, j, 2*k, 7 )]));
698             }
699           }
700         }
701       }
702     }
703   }
704   }
705
706 /* ============================================================================
707   function for load tile file names from input file, create, load and extend
708   tiling map
709   ---------------------------------------------------------------------------- */
710 void loadTiles( FILE *file ) {
711   _TILE_NAMES_ = new char* [ _NO_TILES_ ];
712   for( int i=0; i<_NO_TILES_; i++ ) {
713     _TILE_NAMES_[ i ] = new char [ 100 ];
714     fgetstr( _TILE_NAMES_[ i ], 100, file );
715   }
716
717   fgetstr( _TILING_MAP_FILENAME_, 100, file );
718   map_rows = GetRangeOfFileRows( _TILING_MAP_FILENAME_ );
719   map_columns = GetRangeOfFileColumns( _TILING_MAP_FILENAME_ );
720
721   tile_map = new int* [ map_rows+2 ]; // +2 to create one periodic layer around
722   for( int i=0; i<map_rows+2; i++ ) {
723     tile_map[ i ] = new int [ map_columns+2 ]; // +2 to create one periodic layer around
724   }
725
726   ReadDataToArray( tile_map, _TILING_MAP_FILENAME_ , map_rows+2, map_columns+2 );
727   extendTileMap();
728   }
729
730 /* ============================================================================
731   function for load phases from phase file
732   ---------------------------------------------------------------------------- */
733 void loadPhases( char *file, int n ) {
734   FILE *phase_file = NULL;
735   int phase;
736
737   PhaseMatrix = new double* [ n ];
738   for( int i=0; i<n; i++ ) {
```

```
739       PhaseMatrix[ i ] = new double [ 2 ]; // [ i ][ 0 ] = E; [ i ][ 1 ] = NY
740     }
741
742     OpenFile( &phase_file, file, "rt" );
743     for( int i=0; i<n; i++ ) {
744       fscanf( phase_file, "%d ", &phase );
745       if( phase == i ) {
746         fscanf( phase_file, "%lf %lf ", &PhaseMatrix[ i ][ 0 ], &PhaseMatrix[ i ][ 1 ] );
747       }
748       else {
749         Error( "\nCannot load correct phase, check Phase_file", "");
750         exit( 0 );
751       }
752     }
753     CloseFile( phase_file );
754     }
755
756   /* ============================================================================
757     function for load Load cases from file
758     ---------------------------------------------------------------------------- */
759   void loadLoadCases( char *file, int n ) {
760     FILE *loads_file = NULL;
761
762     Loads = new double* [ n ];
763     for( int i=0; i<n; i++ ) {
764       Loads[ i ] = new double [ 7 ]; // [ i ][e11][e22][e33][e23][e13][e12]
765     }
766
767     OpenFile( &loads_file, file, "rt" );
768     for( int i=0; i<n; i++ ) {
769       fscanf( loads_file, "%lf ", &Loads[ i ][ 0 ] );
770       if( Loads[ i ][ 0 ] == i ) {
771         for( int j=1; j<7; j++ ) {
772           fscanf( loads_file, "%lf", &Loads[ i ][ j ] );
773         }
774       }
775       else {
776         Error( "\nCannot load correct load case, check Load_cases_file", "");
777         exit( 0 );
778       }
779     }
780     CloseFile( loads_file );
781     }
782
783   // ============================================================================
784   void Option_1() {
785     BWBitmap **tile = new BWBitmap* [ _NO_TILES_ ];
786     for( int i=0; i<_NO_TILES_; i++ ) {
787       tile[ i ] = new BWBitmap( _TILE_NAMES_[ i ] );
788     }
789
790     tHeight = tile[ 0 ]->GetHeight();
791     tWidth = tile[ 0 ]->GetWidth();
792     X = 1;
793     Y = 3*tWidth;
794     Z = 3*tHeight;
795     periodx = X;
796     periody = Y;
```

```
797   periodz = Z;
798   scale = X*Y*Z;
799   int dimensions[] = { X, Y, Z };
800   int konverg;
801   double min[ 2 ]={ 0, 0 }; //min eigenvalue
802   double max[ 2 ]={ 0, 0 }; //max eigenvalue
803   TilePhases = create_3D_array_int( _NO_TILES_, tHeight, tWidth );
804   M = create_2D_array_int( Y, Z );
805
806   for( int i=0; i<_NO_TILES_; i++ ) {
807     for( int k=0; k<tHeight; k++ ) {
808       for( int m=0; m<tWidth; m++ ) {
809         TilePhases[ i ][ k ][ m ] = tile[ i ]->GetPixel( m, (( tHeight-1 )-k ));
810       }
811     }
812   }
813
814   printf( "\nCreating plans, arrays and computing reference medium..." );
815   fflush( stdout );
816   S = fftw_alloc_real( X*Y*(2*(Z/2+1))*6 );
817   S_prev = fftw_alloc_real( X*Y*(2*(Z/2+1))*6 );
818   E = fftw_alloc_real( X*Y*(2*(Z/2+1))*6);
819   D = create_4D_array_double( X, Y, Z, 21 );
820   p_S = fftw_plan_many_dft_r2c( 3, dimensions, 6, S, NULL, 6, 1, \
821                                 (fftw_complex*)S, NULL, 6, 1, FFTW_MEASURE );
822   p_E = fftw_plan_many_dft_r2c( 3, dimensions, 6, E, NULL, 6, 1, \
823                                 (fftw_complex*)E, NULL, 6, 1, FFTW_MEASURE );
824   pinv_E = fftw_plan_many_dft_c2r( 3, dimensions, 6, (fftw_complex*)E, \
825                                   NULL, 6, 1, E, NULL, 6, 1, FFTW_MEASURE );
826   Stress_tensors = create_2D_array_double(( map_rows*tHeight ) * \
827                                           ( map_columns*tWidth ), 6 );
828   Strain_tensors = create_2D_array_double(( map_rows*tHeight ) * \
829                                           ( map_columns*tWidth ), 6 );
830   Microstructure = new int[( map_rows*tHeight )*( map_columns*tWidth )];
831
832   for( int load_i=0; load_i<N_loads; load_i++ ) {
833     printf( "\nComputing load case number %d\n", load_i );
834
835     // biggest cycles over each 3x3 tiles from tiling map
836     for( int I=1; I<( map_rows+1 ); I++ ) {
837       for( int J=1; J<( map_columns+1 ); J++ ) {
838         printf( "\nComputing tile[ %d ][ %d ] from tiling map\n", I-1, J-1 );
839         cut3to3Matrix( I, J );
840         copyPhasestoM();
841         computeStiffnessMatrix();
842         get_eig_val( min, max );
843         ii= (( min[ 0 ]+max[ 0 ] )/2 );
844         printf("ref. medium ii=%e\n", ii);
845         fflush(stdout);
846         iter_num=1;
847         konverg=1;
848         initializeStrains( load_i );
849         computeStress();
850         printf( "%d\n", iter_num );
851         store_prev_results();
852
853         while(( konverg!=0 )&&( iter_num<ITERMAX )) {
854           iter_num++;
```

```
855           fftw_execute( p_S );
856           fftw_execute( p_E );
857           printf( "%d\n", iter_num );
858           getFourierStrain();
859           fftw_execute( pinv_E );
860           normalize();
861           computeStress();
862
863           //check at beginning and in every 10th cycle
864           if(( iter_num==2 ) || ( iter_num%10==0 )) {
865             konverg=check_stresses();
866           }
867         }
868         storeData_opt1( I-1, J-1 );
869       }
870     }
871     printVTK( load_i );
872   }
873
874   fftw_destroy_plan( p_S );
875   fftw_destroy_plan( p_E );
876   fftw_destroy_plan( pinv_E );
877   fftw_free( S );
878   fftw_free( S_prev );
879   fftw_free( E );
880   delete_4D_array( D, X, Y, Z );
881   delete_3D_array( TilePhases , _NO_TILES_, tHeight );
882   delete_2D_array( M , Y );
883   delete_2D_array( Stress_tensors , tHeight*tWidth );
884   delete_2D_array( Strain_tensors , tHeight*tWidth );
885   delete [] Microstructure;
886   for( int i=0; i<_NO_TILES_; i++ ) {
887     delete tile[ i ];
888   }
889   }
890
891 // ============================================================================
892 void Option_2() {
893   BWBitmap *original = new BWBitmap( _ORIGINAL_ );
894   tHeight = original->GetHeight( );
895   tWidth = original->GetWidth( );
896   X = 1;
897   Y = tWidth;
898   Z = tHeight;
899   periodx = X;
900   periody = Y;
901   periodz = Z;
902   scale = X*Y*Z;
903
904   int dimensions[] = { X, Y, Z };
905   int konverg;
906   double min[ 2 ]={ 0.0, 0.0 }; //min eigenvalue
907   double max[ 2 ]={ 0.0, 0.0 }; //max eigenvalue
908
909   M = create_2D_array_int( Y, Z );
910   for( int k=0; k<tHeight; k++ ) {
911     for( int m=0; m<tWidth; m++ ) {
912       M[ k ][ m ] = original->GetPixel( m, (( tHeight-1 )-k ));
```

```
913       }
914     }
915
916     printf( "\nCreating plans, arrays and computing reference medium..." );
917     fflush( stdout );
918     S = fftw_alloc_real( X*Y*(2*(Z/2+1))*6 );
919     S_prev = fftw_alloc_real( X*Y*(2*(Z/2+1))*6 );
920     E = fftw_alloc_real( X*Y*(2*(Z/2+1))*6);
921     D = create_4D_array_double( X, Y, Z, 21 );
922     p_S = fftw_plan_many_dft_r2c( 3, dimensions, 6, S, NULL, 6, 1, \
923                                   (fftw_complex*)S, NULL, 6, 1, FFTW_MEASURE );
924     p_E = fftw_plan_many_dft_r2c( 3, dimensions, 6, E, NULL, 6, 1, \
925                                   (fftw_complex*)E, NULL, 6, 1, FFTW_MEASURE );
926     pinv_E = fftw_plan_many_dft_c2r( 3, dimensions, 6, (fftw_complex*)E, \
927                                     NULL, 6, 1, E, NULL, 6, 1, FFTW_MEASURE );
928     Stress_tensors = create_2D_array_double( tHeight* tWidth, 6 );
929     Strain_tensors = create_2D_array_double( tHeight* tWidth, 6 );
930     Microstructure = new int[ tHeight*tWidth ];
931
932     computeStiffnessMatrix(); // computing the stiffness matrix
933     get_eig_val( min, max ); // computing eigen values
934     ii=(( min[ 0 ]+max[ 0 ] )/2 );
935     printf( "ref. medium ii = %e\n", ii );
936     fflush( stdout );
937
938     for( int load_i=0; load_i<N_loads; load_i++ ) {
939       iter_num=1;
940       konverg=1;
941       initializeStrains( load_i );
942       computeStress();
943       printf( "%d\n", iter_num );
944       store_prev_results();
945
946       while(( konverg!=0 )&&( iter_num<ITERMAX )) {
947         iter_num++;
948         fftw_execute( p_S );
949         fftw_execute( p_E );
950         printf( "%d\n", iter_num );
951         getFourierStrain();
952         fftw_execute( pinv_E );
953         normalize();
954         computeStress();
955
956         //check at beginning and in every 10th cycle
957         if(( iter_num==2 ) || ( iter_num%10==0 )) {
958           konverg=check_stresses();
959         }
960       }
961       storeData_opt2();
962       printVTK( load_i );
963     }
964
965     fftw_destroy_plan( p_S );
966     fftw_destroy_plan( p_E );
967     fftw_destroy_plan( pinv_E );
968     fftw_free( S );
969     fftw_free( S_prev );
970     fftw_free( E );
```

```
971   delete_4D_array( D, X, Y, Z );
972   delete_2D_array( M , Y );
973   delete_2D_array( Stress_tensors , tHeight*tWidth );
974   delete_2D_array( Strain_tensors , tHeight*tWidth );
975   delete [] Microstructure;
976   delete original;
977   }
978
979 /* ============================================================================
980   function for load data from input file ( Option 1 )
981   ---------------------------------------------------------------------------- */
982 void loadInput_opt1( ) {
983   FILE *in_file = NULL;
984   char _PHASES_FILENAME_[ 100 ];
985   char _LOADS_FILENAME_[ 100 ];
986   char tmp[ 20 ];
987   int N_phases;
988
989   printf( "\nLoading input..." );
990   OpenFile( &in_file, _INPUT_FILE_, "rt" );
991   fgetstr( tmp, 20, in_file );
992   if ( strcmp( tmp, "Option_1" )) { // checking correct input file
993     Error( "\nProbably wrong input file, check your input file first line, " \
994       "must contain \"Option_1\"\n", "");
995     exit( 0 );
996   }
997
998   //char file_type[ 10 ];
999   // loading type of computation (stress / strain )
1000  //fgetstr( file_type, 10, in_file );
1001  //printf( "\n%s", file_type );
1002
1003  fscanf( in_file, "%d ", &_NO_TILES_ ); // important: must be like this => "%d "
1004  loadTiles( in_file );
1005
1006  fscanf( in_file, "%d " , &N_phases ); // Loading number of phases from input file
1007  fgetstr( _PHASES_FILENAME_, 100, in_file );
1008  loadPhases( _PHASES_FILENAME_, N_phases );
1009
1010  fscanf( in_file, "%d " , &N_loads );
1011  fgetstr( _LOADS_FILENAME_, 100, in_file );
1012  loadLoadCases( _LOADS_FILENAME_, N_loads );
1013
1014  fgetstr( _OUTPUT_FILE_, 100, in_file );
1015  fscanf( in_file, "%lf " , &tolerance );
1016  fscanf( in_file, "%d " , &ITERMAX );
1017  fgetstr( _CREATOR_, 100, in_file );
1018  fgetstr( Stress_name, 100, in_file );
1019  fgetstr( Strain_name, 100, in_file );
1020
1021  CloseFile( in_file );
1022  }
1023
1024 /* ============================================================================
1025   function for load data from input file ( Option 2 )
1026   ---------------------------------------------------------------------------- */
1027 void loadInput_opt2( ) {
1028   FILE *in_file = NULL;
```

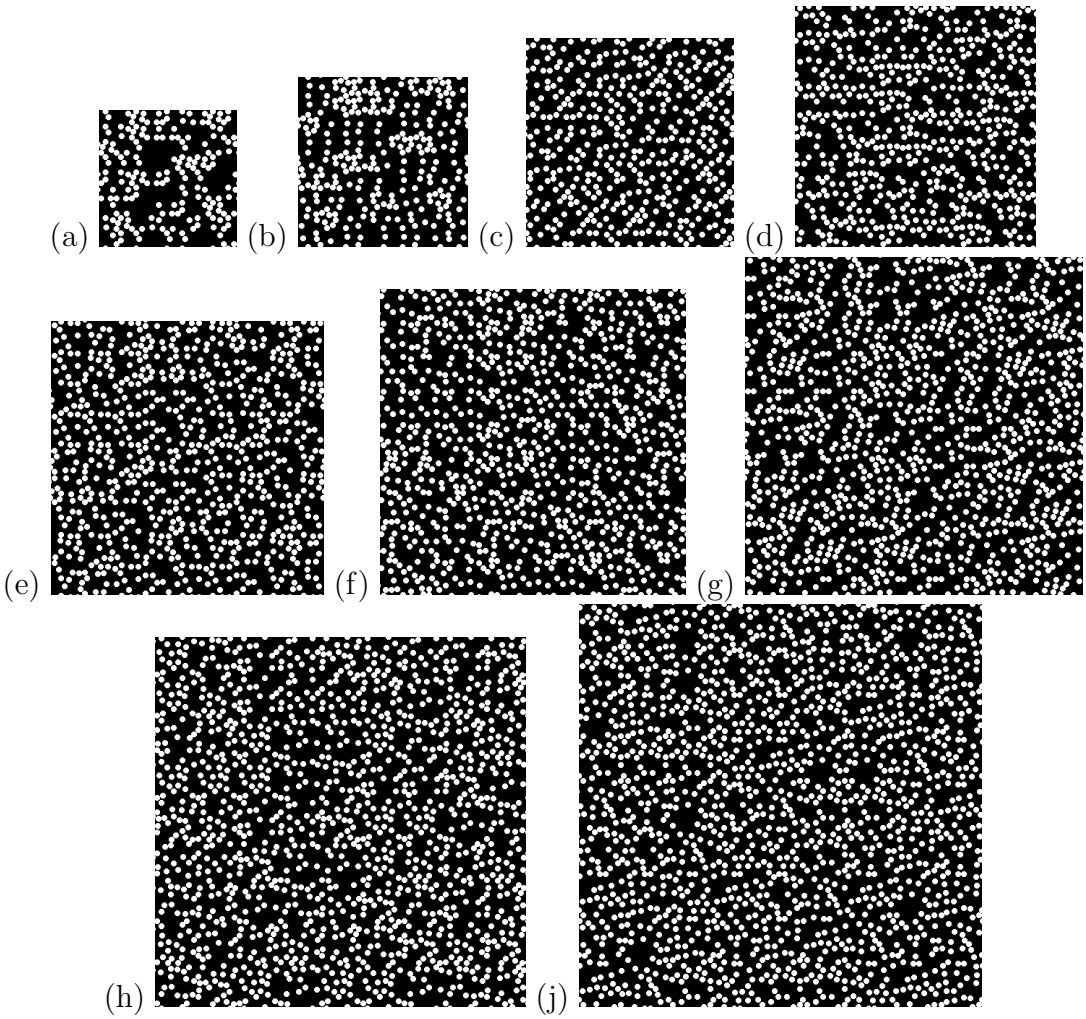
```
1029    char tmp[ 20 ];
1030    char _PHASES_FILENAME_[ 100 ];
1031    char _LOADS_FILENAME_[ 100 ];
1032    int N_phases;
1033
1034    printf( "\nLoading input..." );
1035    OpenFile( &in_file, _INPUT_FILE_, "rt" );
1036    fgetstr( tmp, 20, in_file );
1037    if ( strcmp( tmp, "Option_2" )) { // checking correct input file
1038      Error( "\nProbably wrong input file, check your input file first line, " \
1039        "must contain \"Option_2\"\n", "");
1040      exit( 0 );
1041    }
1042    fgetstr( _ORIGINAL_, 100, in_file );
1043
1044    fscanf( in_file, "%d " , &N_phases );
1045    fgetstr( _PHASES_FILENAME_, 100, in_file );
1046    loadPhases( _PHASES_FILENAME_, N_phases );
1047
1048    fscanf( in_file, "%d " , &N_loads );
1049    fgetstr( _LOADS_FILENAME_, 100, in_file );
1050    loadLoadCases( _LOADS_FILENAME_, N_loads );
1051
1052    fgetstr( _OUTPUT_FILE_, 100, in_file );
1053    fscanf( in_file, "%lf " , &tolerance );
1054    fscanf( in_file, "%d " , &ITERMAX );
1055    fgetstr( _CREATOR_, 100, in_file );
1056    fgetstr( Stress_name, 100, in_file );
1057    fgetstr( Strain_name, 100, in_file );
1058
1059    CloseFile( in_file );
1060    }
1061
1062  /* ============================================================================
1063    function to print HELP for program
1064    ---------------------------------------------------------------------------- */
1065  void printHelp() {
1066    printf( "\nTo run use following syntax:  <executable_file> <option> <input_file>\n"
1067      "-------------------------------------------------------------------------------\n"
1068      "HELP:\n"
1069      "-------------------------------------------------------------------------------\n"
1070      "1 = option to compute with tiles\n"
1071      "needed files:\n"
1072      "\t'input file', 'phases file', 'tiles files', 'tiling map file'\n"
1073      "\t( look in example files )\n"
1074      "2 = option to compute single domain\n"
1075      "needed files:\n"
1076      "\t'input file', 'phases file', 'single domain file'\n"
1077      "\t( look in example files )\n"
1078      "other important facts:\n"
1079      "\t- minimal dimensions to compute are 3x3 pixels\n"
1080      "\t- file containing tile map, MUST end with empty line\n"
1081      "-------------------------------------------------------------------------------\n"
1082      );
1083    }
1084
1085  /* ============================================================================
1086    Main function
```

```
1087    argv[ 0 ] = executable file
1088    argv[ 1 ] = option ( 1 = compute with tiles; 2 = compute one domain )
1089    argv[ 2 ] = input file
1090    ------------------------------------------------------------------------------ */
1091  int main( int argc, char *argv[] ){
1092
1093    if( argc != 3 ){
1094      printHelp();
1095      return 0;
1096    }
1097
1098    strcpy( _INPUT_FILE_, argv[ 2 ] );
1099
1100    if( strcmp( argv[ 1 ], "1" ) == 0 ) {
1101      printf( "\nOption 1:\n" );
1102      loadInput_opt1();
1103      Option_1();
1104      printf( "\nSuccessfully completed!\n\a" );
1105
1106      return( 1 );
1107    }
1108
1109    else if( strcmp( argv[ 1 ], "2" ) == 0 ) {
1110      printf( "Option 2:\n" );
1111      loadInput_opt2();
1112      Option_2();
1113      printf( "\nSuccessfully completed!\n\a" );
1114      return( 1 );
1115    }
1116
1117    else {
1118      printf( "\nWrong or missing option.\n\n" );
1119      printHelp();
1120      return( 0 );
1121    }
1122    }
```

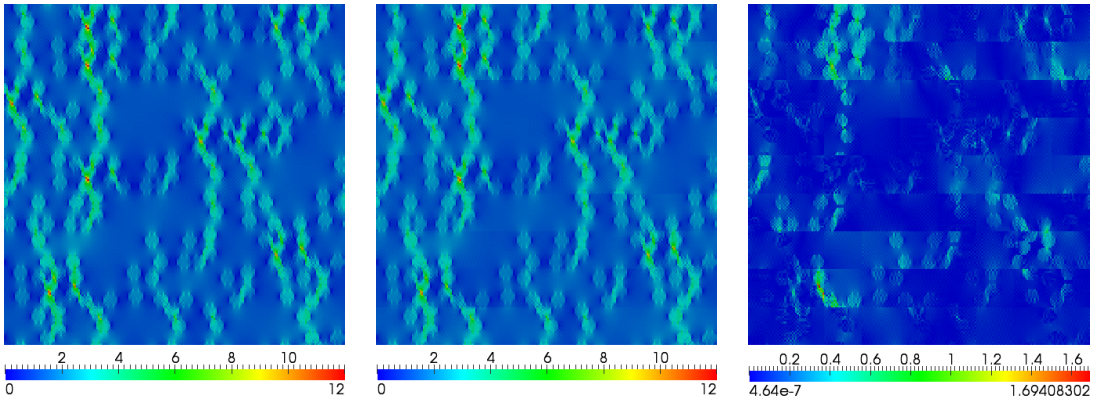
Příloha B

Obrázky

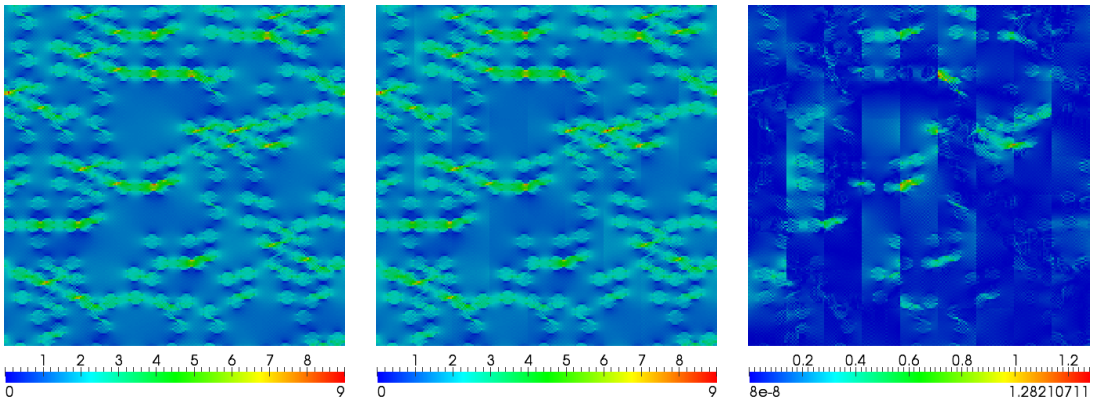
Příloha obsahuje všechny obrázky a výsledky získané při tvorbě této diplomové práce.



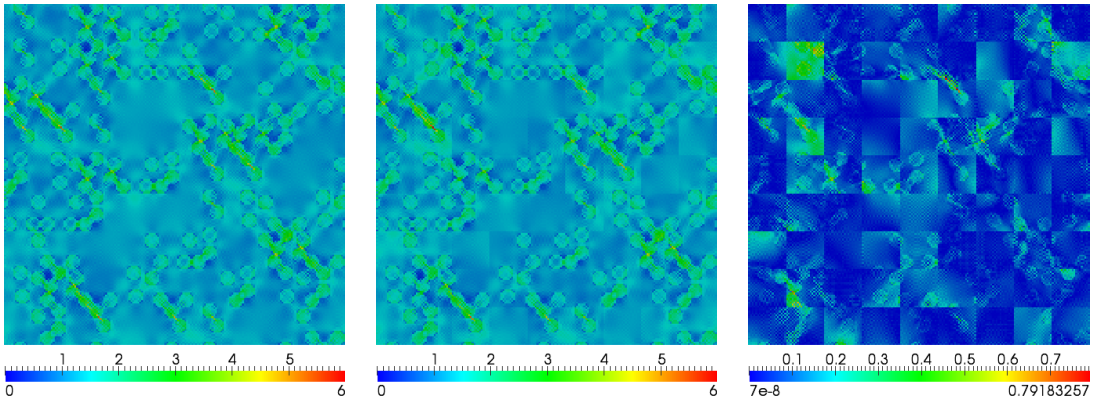
Obrázek B.1: Zrekonstruované mikrostruktury: (a) W8/2-2-010, (b) W8/2-2-017, (c) W8/2-2-027, (d) W8/2-2-038, (e) W8/2-2-052, (f) W8/2-2-067, (g) W8/2-2-084, (h) W8/2-2-104, (j) W8/2-2-125



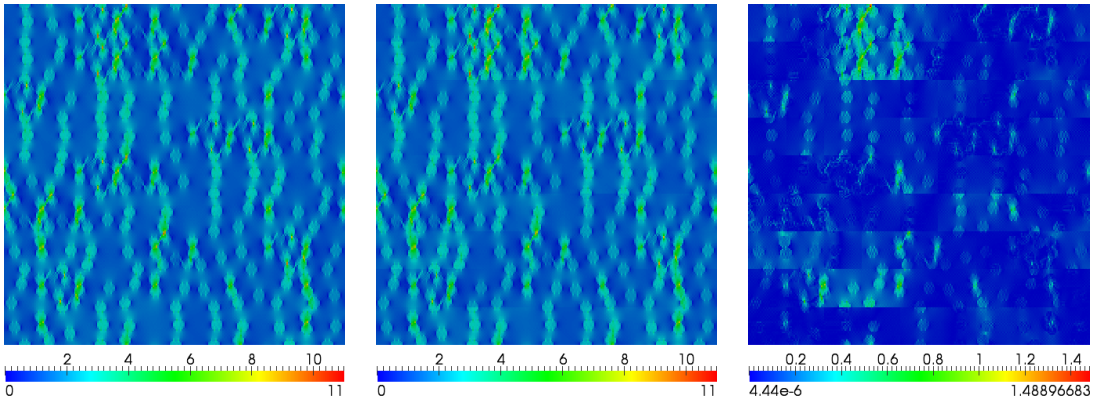
Obrázek B.2: Množina W8/2-2-010, složka napětí σ_y



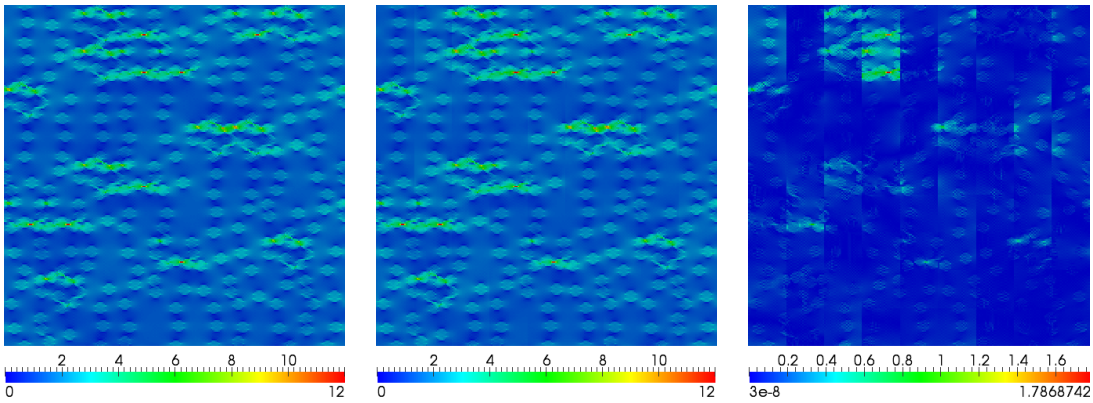
Obrázek B.3: Množina W8/2-2-010, složka napětí σ_z



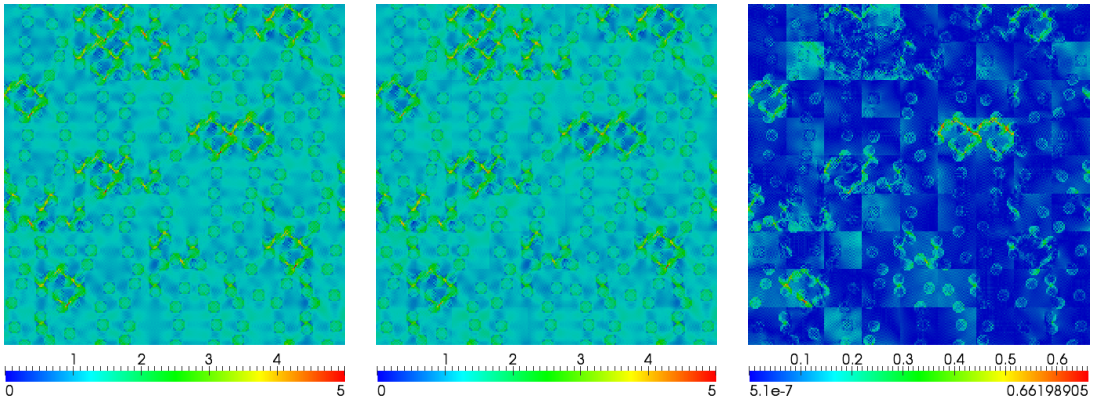
Obrázek B.4: Množina W8/2-2-010, složka napětí τ_{yz}



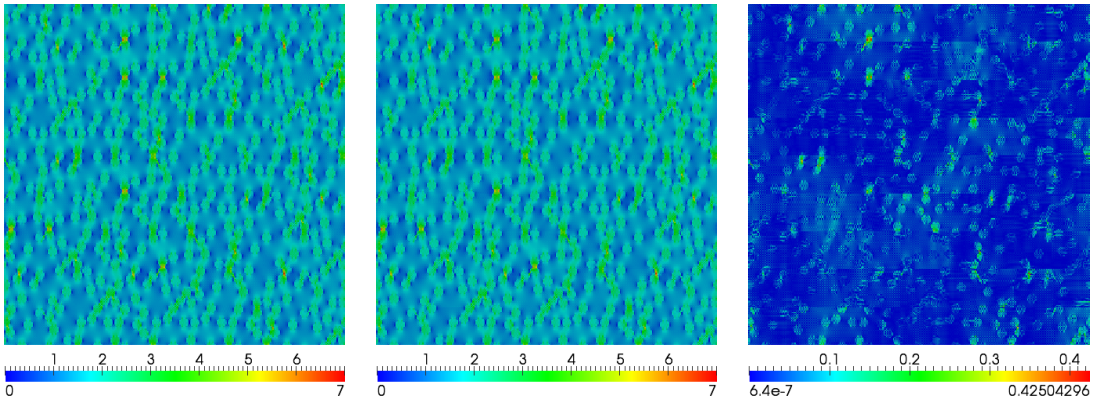
Obrázek B.5: Množina W8/2-2-017, složka napětí σ_y



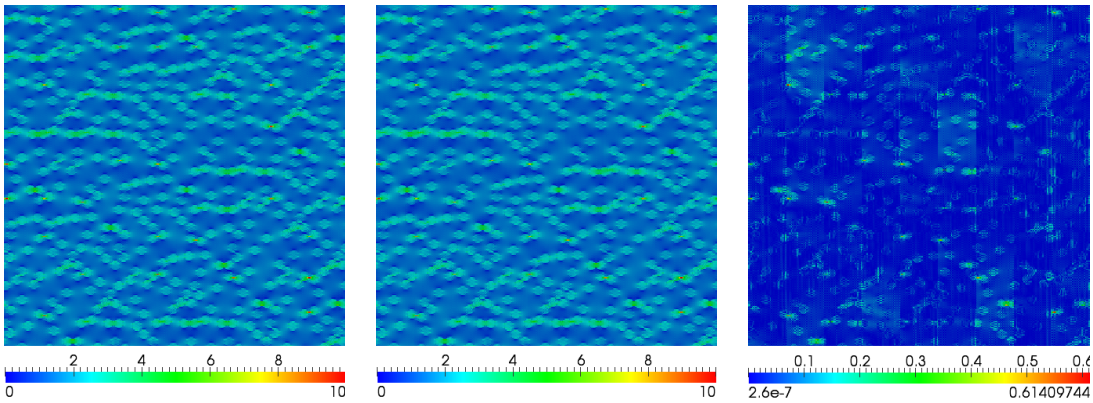
Obrázek B.6: Množina W8/2-2-017, složka napětí σ_z



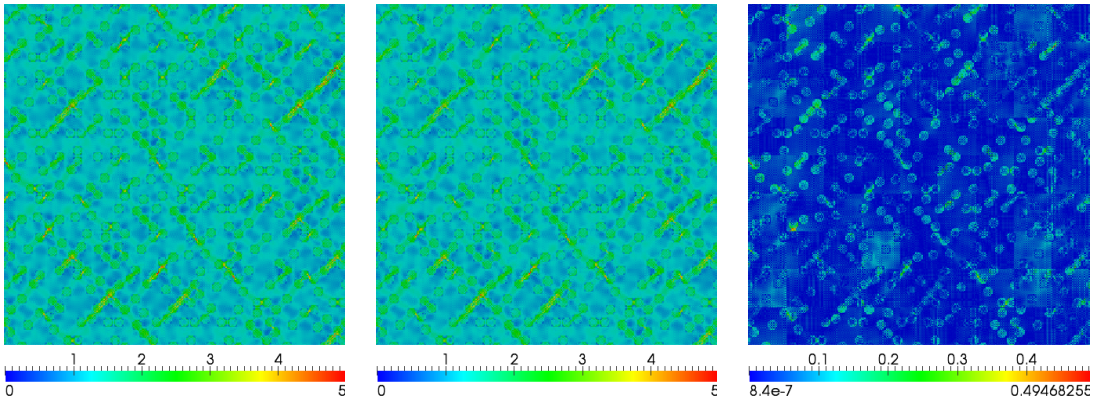
Obrázek B.7: Množina W8/2-2-017, složka napětí τ_{yz}



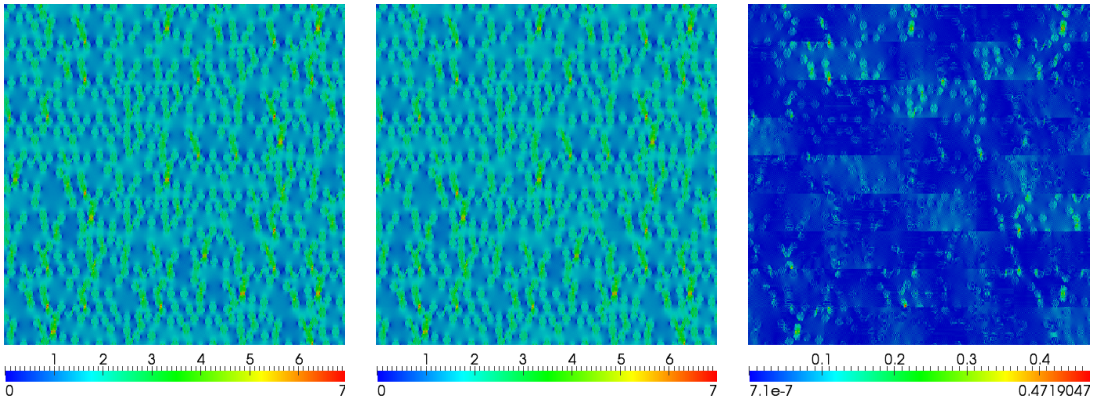
Obrázek B.8: Množina W8/2-2-027, složka napětí σ_y



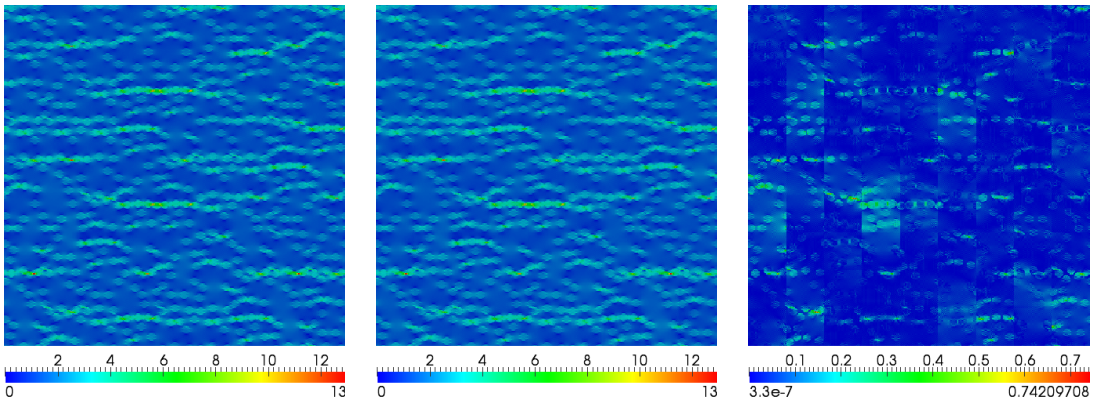
Obrázek B.9: Množina W8/2-2-027, složka napětí σ_z



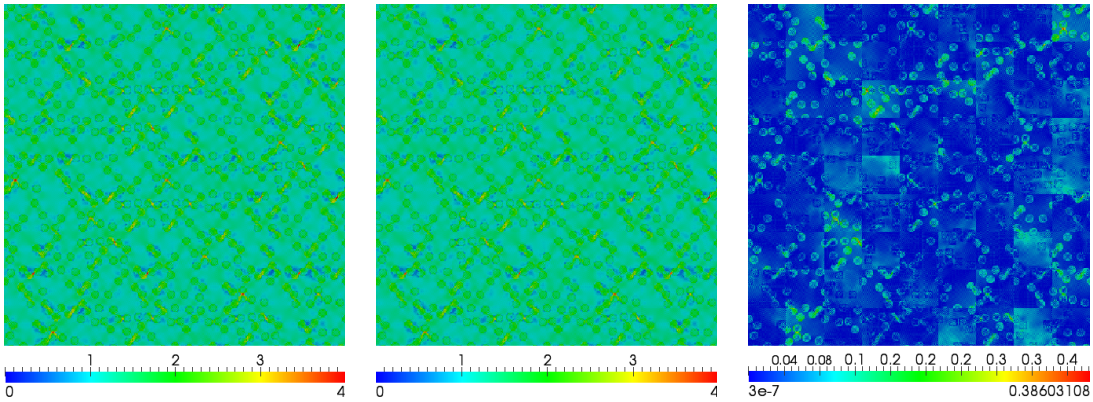
Obrázek B.10: Množina W8/2-2-027, složka napětí τ_{yz}



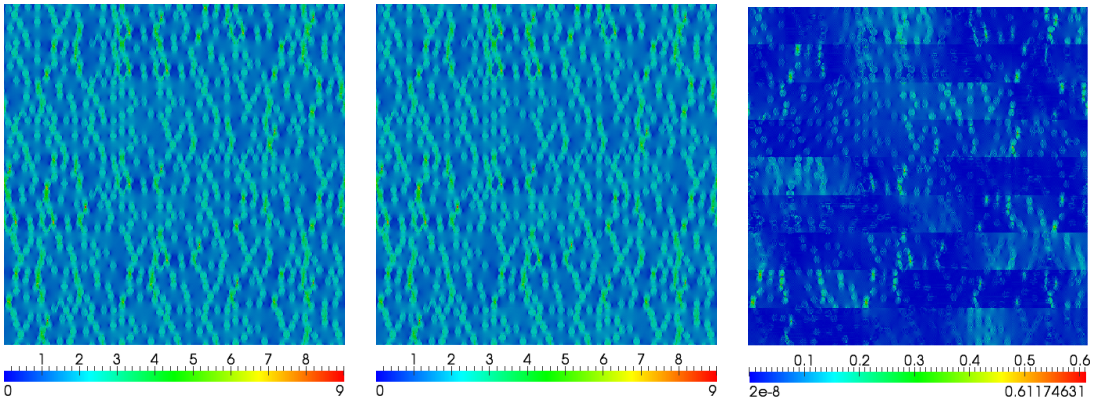
Obrázek B.11: Množina W8/2-2-038, složka napětí σ_y



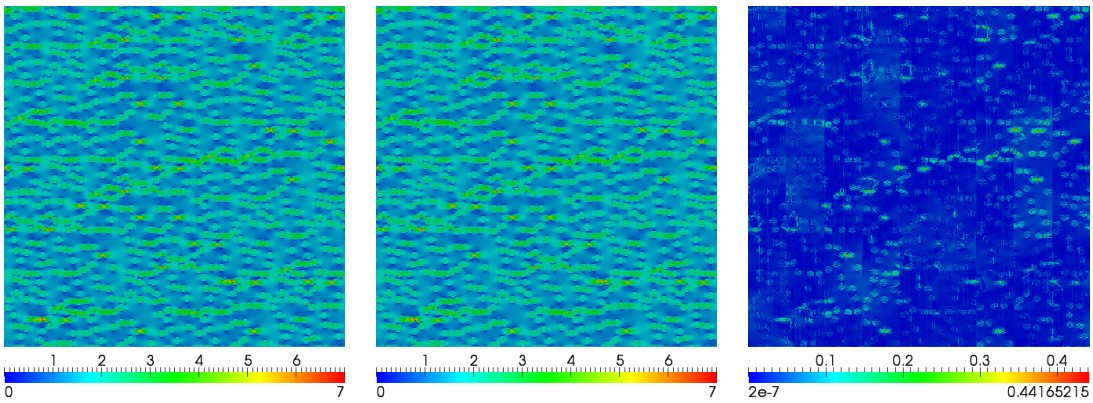
Obrázek B.12: Množina W8/2-2-038, složka napětí σ_z



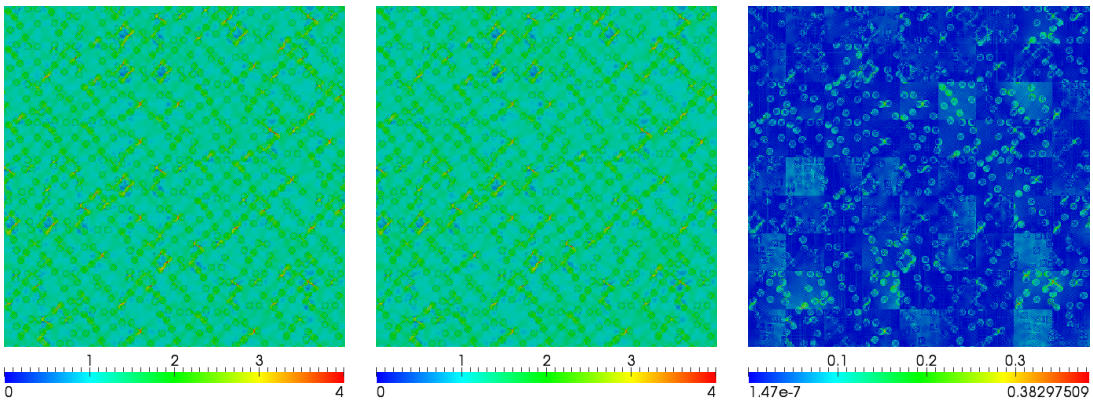
Obrázek B.13: Množina W8/2-2-038, složka napětí τ_{yz}



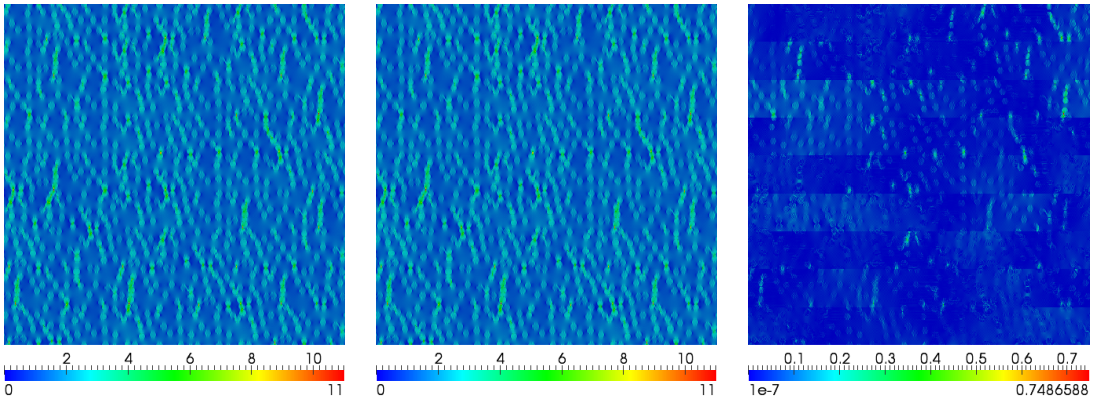
Obrázek B.14: Množina W8/2-2-052, složka napětí σ_y



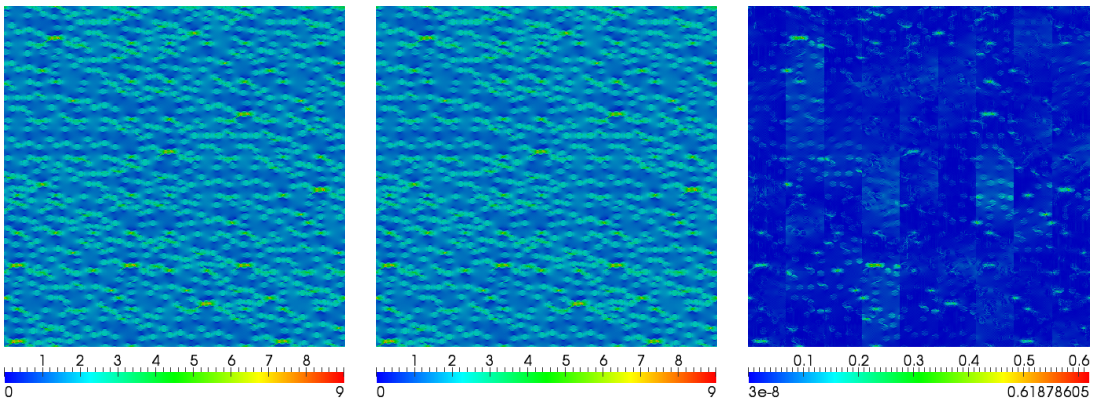
Obrázek B.15: Množina W8/2-2-052, složka napětí σ_z



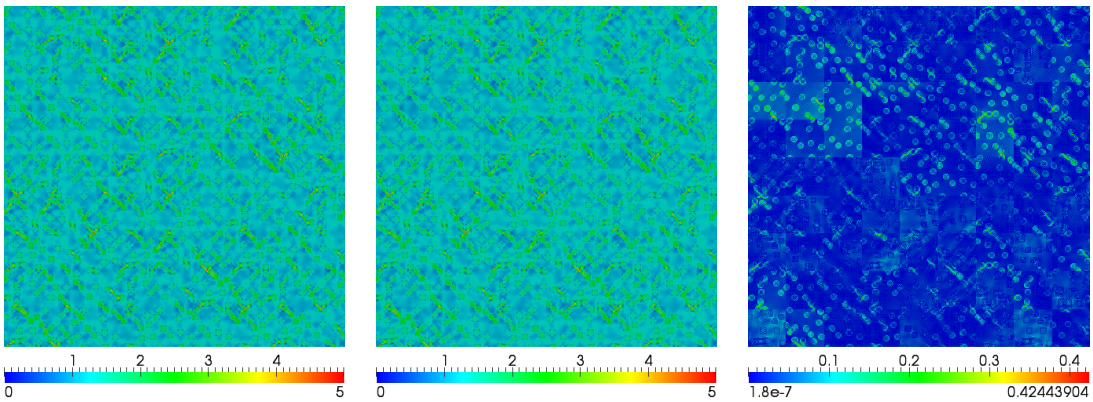
Obrázek B.16: Množina W8/2-2-052, složka napětí τ_{yz}



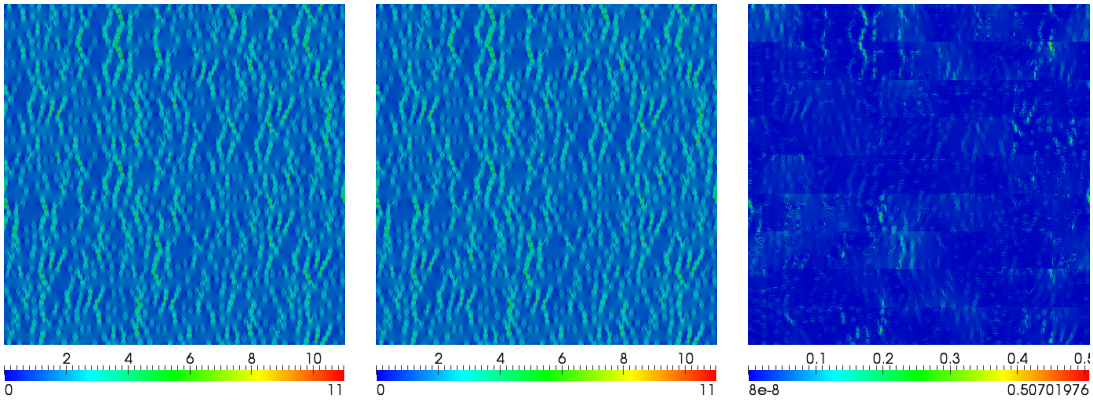
Obrázek B.17: Množina W8/2-2-067, složka napětí σ_y



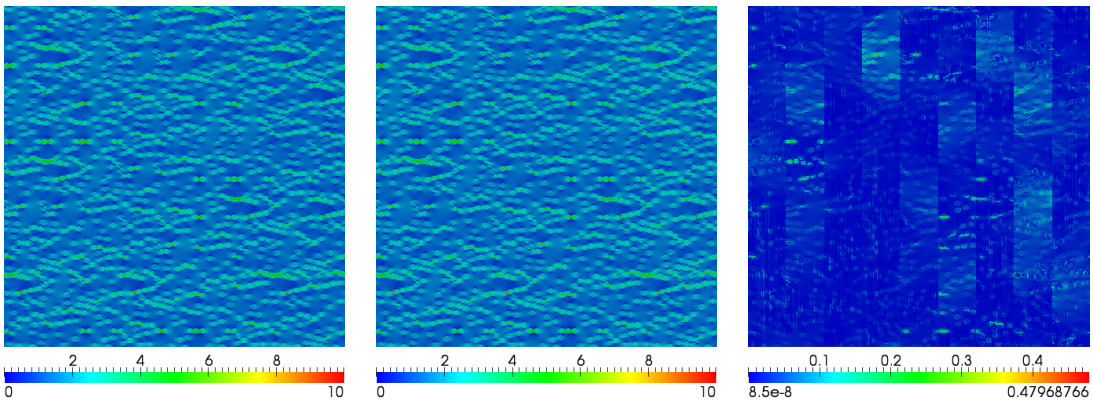
Obrázek B.18: Množina W8/2-2-067, složka napětí σ_z



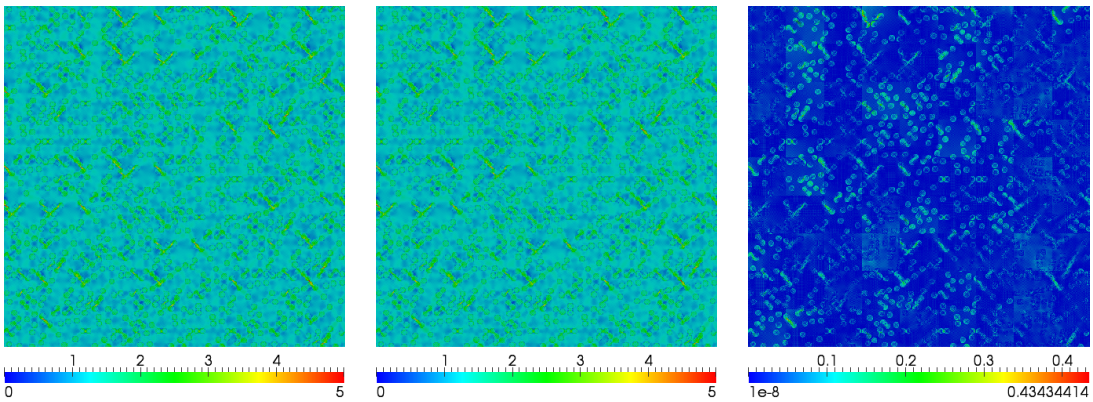
Obrázek B.19: Množina W8/2-2-067, složka napětí τ_{yz}



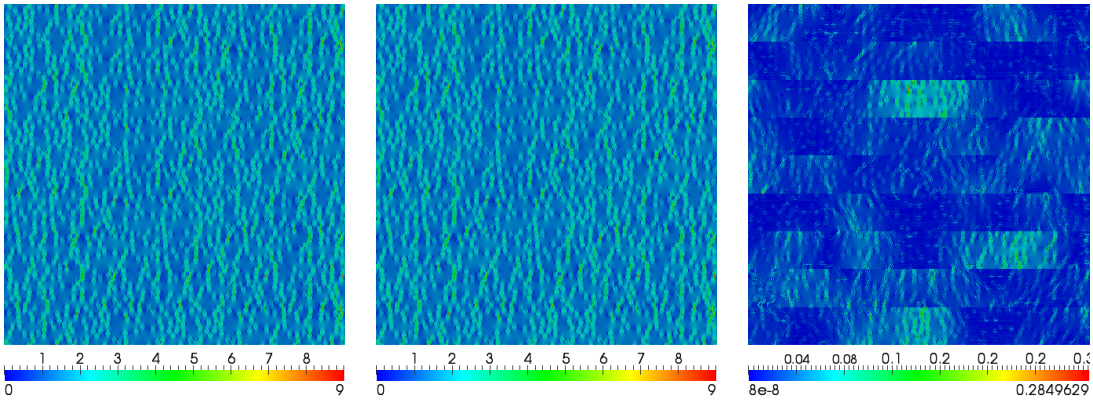
Obrázek B.20: Množina W8/2-2-084, složka napětí σ_y



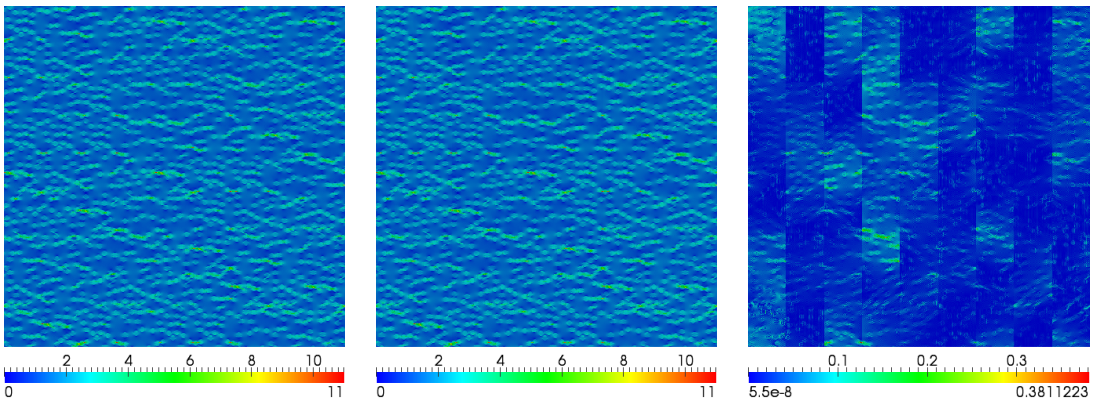
Obrázek B.21: Množina W8/2-2-084, složka napětí σ_z



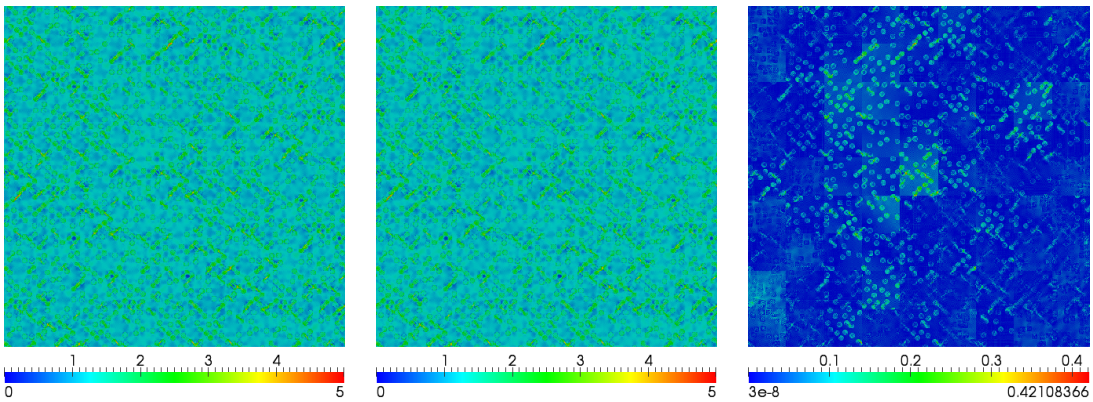
Obrázek B.22: Množina W8/2-2-084, složka napětí τ_{yz}



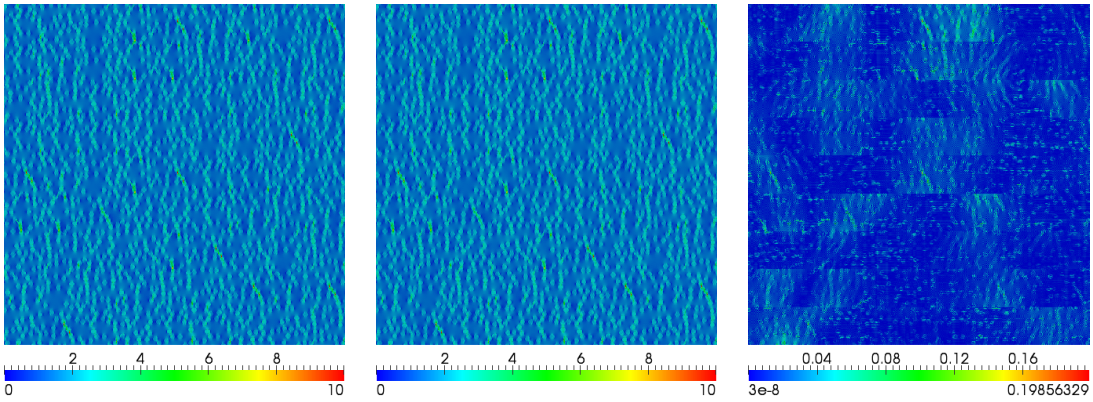
Obrázek B.23: Množina W8/2-2-104, složka napětí σ_y



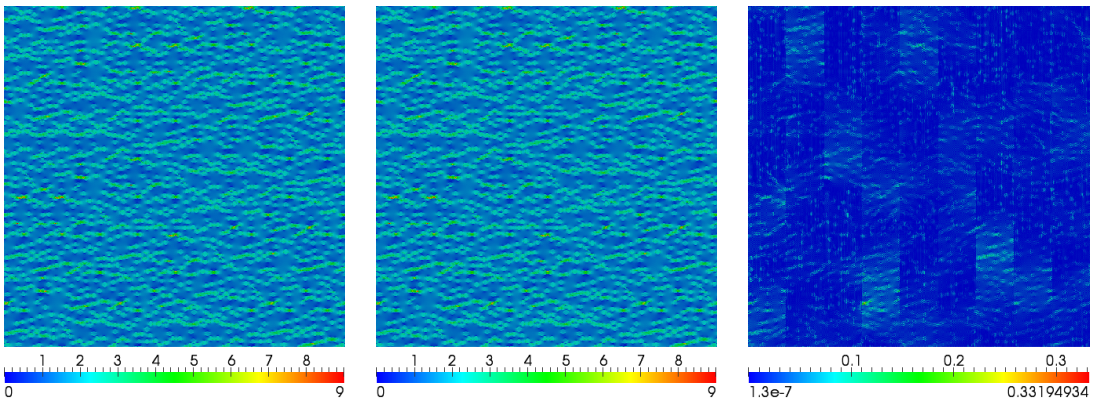
Obrázek B.24: Množina W8/2-2-104, složka napětí σ_z



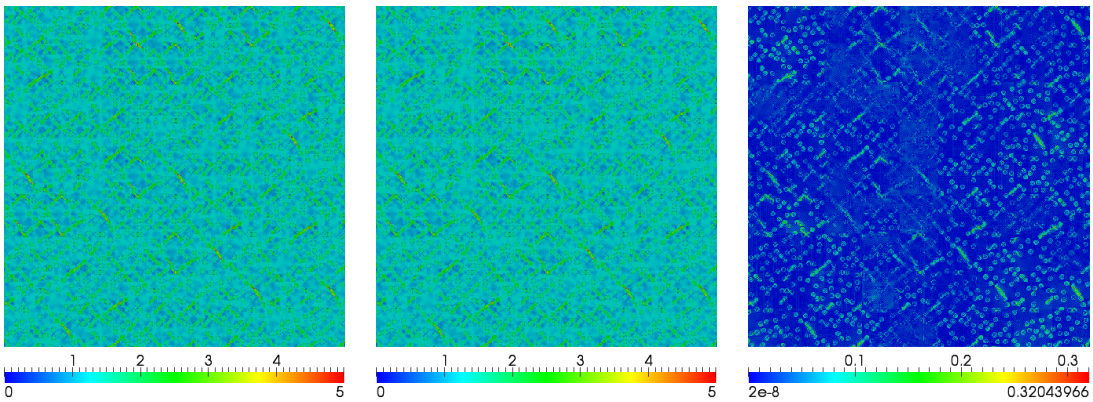
Obrázek B.25: Množina W8/2-2-104, složka napětí τ_{yz}



Obrázek B.26: Množina W8/2-2-125, složka napětí σ_y



Obrázek B.27: Množina W8/2-2-125, složka napětí σ_z



Obrázek B.28: Množina W8/2-2-125, složka napětí τ_{yz}

Příloha C

Seznam použitého softwaru

Následující seznam zahrnuje hlavní použité programy při tvorbě této práce, stejně tak při programování programu popsaného v kapitole 5 a při výpočtech.

- operační systém - `Microsoft Windows 7 (64bit)`
- editor kódu - `Sublime Text 2.0.1`
- programovací prostředí - `Open Watcom 1.9`
- editor jazyka LaTeX- `WinEdt 7.0 a˜MiKTex 2.9`
- prohlížeč souborů typu VTK - `Paraview 3.14`
- kancelářský balík - `Microsoft Office 2007`

Příloha D

Obsah přiloženého CD

Na přiloženém CD jsou k dispozici tyto soubory.

- `DP_2012_Zrubek.pdf` - diplomová práce ve formátu PDF
- `main.cpp` - hlavní soubor programu
- `libfftw3-3.dll` - dynamická knihovna FFTW (určeno pro MS Windows)
- `libw.lib` - statická knihovna FFTW (určeno pro MS Windows)
- `fftw3.h` - hlavičkový soubor knihovny FFTW
- `bmp.cpp` - knihovna pro práci s binárními bitmapami
- `bmp.h` - hlavičkový soubor knihovny pro práci s binárními bitmapami
- `share.h` - hlavičkový soubor obsahující potřebná makra
- `postWangTiles.cpp` - knihovna pro výstup dat do souborů typu VTK
- `postWangTiles.h` - hlavičkový soubor knihovny pro výstup dat do souborů typu VTK
- `primaryFunctions.cpp` - knihovna základních funkcí
- `primaryFunctions.h` - hlavičkový soubor knihovny základních funkcí
- `input_file_opt1.txt` - vzorový vstupní soubor pro variantu 1
- `input_file_opt2.txt` - vzorový vstupní soubor pro variantu 2
- `load_cases.txt` - vzorový soubor seznamu zatěžovacích stavů
- `phases.txt` - vzorový soubor seznamu fází
- `tiling_map.txt` - vzorový soubor mapy dláždění