Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Modul pro transformaci dat z výpočetní do grafické vrstvy

Plzeň, 2013 Jan Zeman

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne .....

Abstract

Module for the data transformation from the computation into the graphic layer

The aim of this thesis is to extend the current program for scientific computation with a repository. The repository should enable the user to store and visualize the whole computation process and to run the solution again, starting from any iteration, possibly with different parameters. By design, we strive the repository to exhaust only the minimum of the resources. Then, we explore a couple of programs for the visualization of scientific data to find out the right file format to export a mesh to. The user should be able to extract data from this mesh and to create an animation from the series of them.

Obsah

1 Úvod

Dynamika tekutin je popsána rovnicemi, které nelze řešit přesně, pouze je za pomoci numerické matematiky aproximovat. K tomu nám jsou k dispozici standardní programy (např. OpenFoam nebo FreeFem++, používající metodu nejmenších prvků), které nabízejí širokou funkcionalitu pro složité výpočty a jsou určeny na vědecká pracoviště. Program NTMSH, vyvinutý na Katedře informatiky UJEP v Ústí nad Labem, má pro potřeby univerzitního výzkumu omezený rozsah jen na výpočet rovnic dynamiky tekutin pomocí metody konečných objemů. Je určený pro řešení speciálnějších problémů.

Cílem naší práce bylo doplnit program o úložiště a modul pro vizualizaci extrahovaných dat. Motivací k tomu byly poměrně dlouhé výpočty, jejichž průběh nebyl nikde zaznamenán, a při přerušení musely tedy být počítány znovu od začátku.

V teoretické části čtenáři nejprve představíme rovnice proudění a metodu konečných objemů, pomocí které je řešíme. Poté, co vysvětíme matematicko-fyzikální podklad programu, seznámíme čtenáře s jeho současným stavem. Prozkoumáme dále programy, které umožní vizualizovat výsledky, vypočtené ze zmíněných rovnic, a zjistíme, s jakými formáty souborů operují.

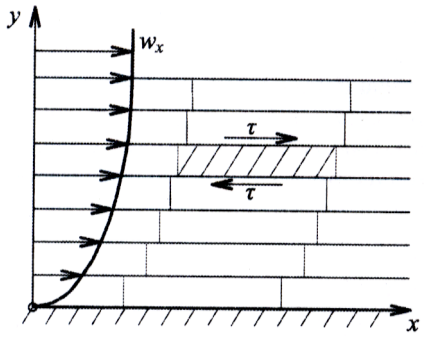
V realizační části poté navrhneme úložiště, které bude umožňovat ukládat a načítat řešení v konkrétní iteraci či konkrétním času výpočtu. Pomocí něj bude navíc možno vytvářet animace výpočtu či libovolně změnit jeho parametry za běhu. Popíšeme vztah úložiště k vizualizační vrstvě a uvedeme si příklady, jak lze data z úložiště podle libosti transformovat. Vybereme nejvhodnější formát souboru pro vizualizaci dat a otestujeme, zda jsou splněny všechny stanovené případy užití. Modul použijeme na reálných problémech.

2 Teoretická část

2.1 Teorie dynamiky tekutin

Tekutina je látka, jejíž částice se navzájem snadno přemisťují. U kapaliny jsou částice stále ještě vázané podobně, jako u pevných látek, mohou se však ze své základní pozice vychýlit. U plynů již putují molekuly do prostoru a je mezi nimi větší střední vzdálenost. Nejprve objasníme pojmy *vazkost* a *stlačitelnost*, se kterými budeme pracovat.

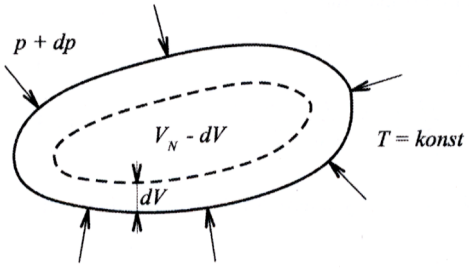
- *Vazkost* je vlastnost tekutiny, která zmenšuje rozdíl mezi ní a okolím. Představme si proudění, složené z vrstev na základě jejich rychlosti w_x (viz obr. 1). Molekuly blíže k okraji ($y \to 0$) jsou pomalejší než dál od něj. Rychlejší vrstva zpomaluje sousední pomalejší a zároveň je urychlována vrstvou ještě rychlejší dál od okraje. Působí smykové tření τ, které závisí na koeficientech vazkosti tekutiny.



Obrázek 1: Pricip vazkých sil

- *Stlačitelnost* je pak schopnost molekul tekutiny změnit svou polohu a přiblížit se k sobě. Nestlačitelné tekutiny mají konstantní hustotu. U stlačitelných závisí na tlaku p. Při jeho zvýšení o dp se vytknutý objem tekutiny V_N zmenší o dV (viz obr. 2). Platí, že plyny jsou stlačitelné, kapaliny velmi málo. Stlačitelnost kapalin se projevuje až při velkých rozdílech tlaků (viz [2, str. 5]).

Ideální kapalina je nevazká a nestlačitelná, ideální plyn potom nevazký a stlačitelný. Tatáž tekutina se však při pomalých rychlostech může chovat jako vazká a při vysokých jako nevazká [2, str. 35]. Poznamenejme, že tekutina vždy vykazuje jistou míru vazkosti. Pokud hovoříme o nevazkém proudění tekutiny, je její koeficient vazkosti tak malý, že jej lze v rovnicích zanedbat. Stlačitelné proudění tekutiny popisují Navier-Stokesovy rovnice (1), které



Obrázek 2: Stlačitelnost tekutin

pak pro nevazká proudění zjednodušují rovnice Eulerovy (2) zanedbáním některých jejich členů.

2.1.1 Navier-Stokesovy rovnice

Zavedeme stavový vektor $\boldsymbol{w}$, který nám bude udávat hledané fyzikální veličiny: hustotu ρ, hybnosti ve směrech os $(\rho v_1, \rho v_2, \rho v_3)$ a celkovou energii E. Tyto jeho složky se nazývají *konzervativní proměnné*:

$$\boldsymbol{w} = (w_1, w_2, w_3, w_4, w_5) = (\rho, \rho v_1, \rho v_2, \rho v_3, E)^T$$

Navier-Stokesovy rovnice pro stlačitelné proudění lze nyní za pomoci stavového vektoru $\boldsymbol{w}$ zapsat takto:

$$\frac{\partial \boldsymbol{w}}{\partial t} + \sum_{s=1}^{3} \frac{\partial \boldsymbol{f}_s(\boldsymbol{w})}{\partial x_s} = \boldsymbol{F}(\boldsymbol{w}) + \sum_{s=1}^{3} \frac{\partial \boldsymbol{R}_s(\boldsymbol{w}, \nabla \boldsymbol{w})}{\partial x_s} \tag{1}$$

$\boldsymbol{f}_s$ bude označovat nevazké toky:

$$\boldsymbol{f}_s(\boldsymbol{w}) = \begin{pmatrix} \rho v_s \\ \rho v_1 v_s + \delta_{1s} p \\ \rho v_2 v_s + \delta_{2s} p \\ \rho v_3 v_s + \delta_{3s} p \\ (E + p) v_s \end{pmatrix} \quad \boldsymbol{F}(\boldsymbol{w}) = \begin{pmatrix} 0 \\ \rho f_1 \\ \rho f_2 \\ \rho f_3 \\ \rho \boldsymbol{f} \cdot \boldsymbol{v} + \rho q \end{pmatrix}$$

$\boldsymbol{R}_s$ potom toky vazké:

$$\boldsymbol{R}_s = \begin{pmatrix} 0 \\ \tau'_{s1} \\ \tau'_{s2} \\ \tau'_{s3} \\ (\tau' \boldsymbol{v})_s + k \frac{\partial \theta}{\partial x_s} \end{pmatrix},$$

kde symbol p udává tlak, τ' představuje tenzor napětí a τ'_{ij} jeho složky, Θ absolutní teplotu, k je konstanta tepelné vodivosti. Pro pozorovanou oblast vyplněnou tekutinou budeme používat symbol Ω, podrobněji např. v [1, str. 10].

2.1.2 Eulerovy rovnice

Pokud předpokládáme adiabatické proudění, tj. zanedbáme přenos tepla, a navíc půjde o proudění nevazké, dostaneme nelineární systém *Eulerových rovnic* (odvození viz [1, str. 21]) :

$$\frac{\partial \boldsymbol{w}}{\partial t} + \sum_{s=1}^{3} \frac{\partial \boldsymbol{f}_s(\boldsymbol{w})}{\partial x_s} = 0 \qquad \text{v } Q_T \tag{2}$$

s počáteční podmínkou

$$\boldsymbol{w}(x,0) = \boldsymbol{w}^0(x), \qquad x \in \Omega$$

a okrajovou podmínkou

$$B(\boldsymbol{w}(x,t)) = 0 \qquad \text{pro } (x,t) \in \partial\Omega \times (0,T),$$

kde $\partial\Omega$ je hranice oblasti, $Q_T = \Omega \times (0,T)$ je časoprostorový válec, $\boldsymbol{w}^0$ je daná vektorová funkce a B je vhodný operátor, popisující okrajové podmínky.

2.1.3 Mělká voda

Pro rovnice, popisující nestlačitelné proudění v mělké vodě, zavádíme

$$\boldsymbol{w} = (h, hv_1, hv_2)^T,$$

kde h je výška hladiny. Pracuje se tedy ve dvou rozměrech. Výsledné rovnice jsou podobného typu, jako rovnice Eulerovy, ale liší se v parametru $\boldsymbol{f}_s(\boldsymbol{w})$. Jejich formální zápis však bude analogický:

$$\frac{\partial \boldsymbol{w}}{\partial t} + \sum_{s=1}^{2} \frac{\partial \boldsymbol{f}_s(\boldsymbol{w})}{\partial x_s} = s(\boldsymbol{x}, \boldsymbol{w}), \tag{3}$$

kde $s(\boldsymbol{x}, \boldsymbol{w})$ je zdrojový člen.

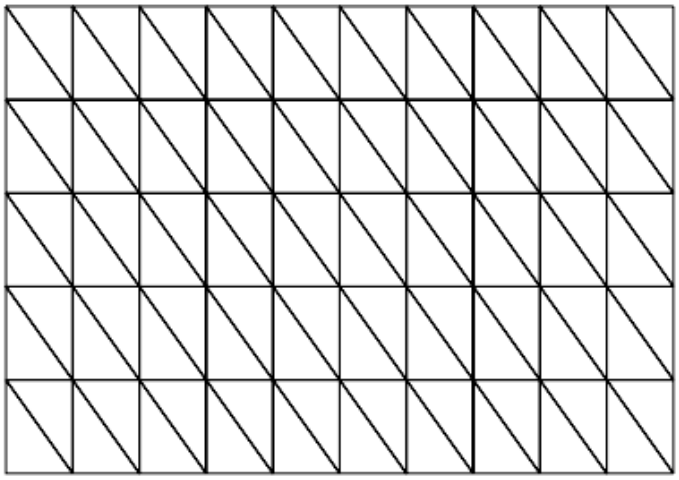
2.2 Metoda konečných objemů

Nyní potřebujeme najít způsob, jak z daných rovnic vypočítat vektor $\boldsymbol{w}$ v daném místě a čase. My jsme k tomu z existujících postupů vybrali metodu konečných objemů. Spočívá v tom, že oblast Ω budeme modelovat jako síť, na jejichž dílech, budeme předpokládat po částech konstantní řešení. Tyto díly budeme nazývat konečnými objemy.

2.2.1 Síť konečných objemů

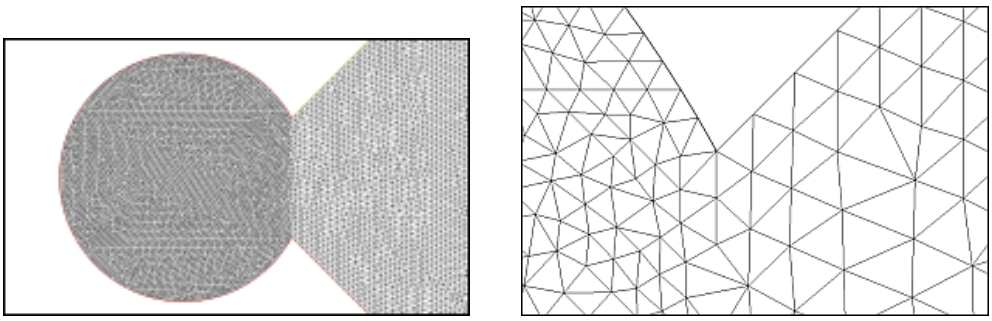
Pod pojmem síť budeme rozumět polygonální diskretizaci stanovené oblasti na jednotlivé díly. Těm říkáme konečné objemy, značíme D_i. Narazili jsme tedy již na druhý termín, kde může být čtenář zmaten názvoslovím. V našem kontextu nebude síť znamenat počítačovou síť, nýbrž geometrický objekt. Objem nebude znamenat fyzikální veličinu, nýbrž jeden díl sítě. Síť může být jednorozměrná (intervaly), dvojrozměrná, nebo prostorová. V případě dvojrozměrné může být např. tvořena trojúhelníky nebo čtyřúhelníky nebo kombinací obou těchto objektů.

Strukturovaná síť je taková, kde se v každém jejím uzlu stýká stejný počet konečných objemů. Typickým příkladem je obyčejná mřížka (viz obr. 3).



Obrázek 3: Strukturovaná síť

V našem případě síť takto pravidelný tvar obecně mít nemusí, pracujeme na nestrukturované dvojrozměrné síti, a musíme tak ukládat i pozice všech jejích uzlů (viz příklad na obr. 4).



Obrázek 4: Nestrukturovaná síť a její detail

2.2.2 Schéma výpočtu

Zaveďme dělení času $t_1, t_2, ...$ a časový krok $\tau_k = t_{k+1} - t_k$. Potom bude vektor $\boldsymbol{w}_i^k$ přibližné řešení na konečném objemu D_i v čase t_k. Stav v dalším čase tedy počítáme na základě stavu v předcházejícím. Metoda se zastaví buď po stanoveném čase, anebo pokud se řešení ze současné a předchozí iterace liší o dostatečně malou hodnotu, tedy pokud nastane tzv. *stacionární stav*.

Počáteční podmínku z rovnice 2 vypočítáme pro každý konečný objem integrací stanovené funkce $\boldsymbol{w}^0(x)$ přes jeho plochu.

$$\boldsymbol{w}_i^0 = \frac{1}{|D_i|} \int_{D_i} \boldsymbol{w}^0(x) dx, \qquad x \in \Omega \tag{4}$$

Metoda má potom takovéto schéma (odvození viz [1, str. 37]):

$$\boldsymbol{w}_i^{k+1} = \boldsymbol{w}_i^k - \frac{\tau_k}{|D_i|} \sum_{j \in S(i)} \boldsymbol{H}(\boldsymbol{w}_i^k, \boldsymbol{w}_j^k, n_{ij}) |\Gamma_{ij}|, \qquad D_i \in \Omega_h, t_k \in [0, T), \tag{5}$$

kde $|D_i|$ je plocha konečného objemu, $S(i)$ je množina indexů sousedních konečných objemů k objemu D_i, $|\Gamma_{ij}|$ je délka společné hranice dvou konečných objemů. $\boldsymbol{H}$ aproximuje tok fyzikálních veličin mezi D_i a D_j. Konstrukce numerického toku je matematicky složitá a stojí mimo záměr tohoto textu. Zmiňme pouze, že náš program umožňuje pracovat s různými typy numerických toků, např. *Vijaysundaramovým* či se *Stegerův-Warmingovým* a pro bližší informace o nich odkážeme čtenáře na ([1, str. 41]).

Počítaná data, tedy vektor $\boldsymbol{w}$, příslušejí konečnému objemu, tedy na ploše, kterou vymezují uzly sítě. V případě, že bychom potřebovali hodnoty dat přímo na uzlech, museli bychom použít transformační metody.

2.2.3 Transformace a extrakce dat

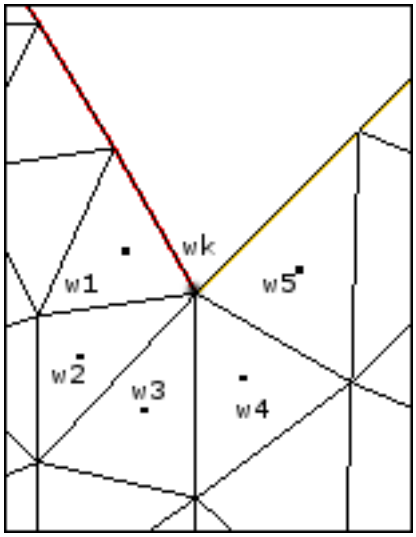
V každém uzlu sítě se stýká několik konečných objemů, na kterých jsou data. Transformaci dat na uzly provádíme dvěma způsoby:

1. *Váženým průměrem*

   Průměrujeme data všech konečných objemů, do kterých uzel patří. Vahou bude přitom jejich plocha.

$$\boldsymbol{w}_k = \frac{\sum_{j \in T(k)} \boldsymbol{w}_j |D_j|}{\sum_{j \in T(k)} |D_j|},$$

   kde $\boldsymbol{w}_k$ je rekonstruované řešení na uzlu s indexem k a $T(k)$ je množina všech konečných objemů, do kterých uzel k přísluší (viz obr. 5).



Obrázek 5: Rekonstrukce řešení

2. *Metodou nejmenších čtverců*

   Pro každý uzel k o souřadnicích (x_k, y_k) chceme nalézt řešení

$$\boldsymbol{w}_k = \boldsymbol{a}_k x_k + \boldsymbol{b}_k y_k + \boldsymbol{c}_k.$$

   Pro každý uzel vycházíme ze známých dat okolních konečných objemů. Vytvoříme takovou síť bodů v prostoru, jejichž první dvě souřadnice budou tvořit souřadnice středů okolních konečných objemů a poslední třetí souřadnici řešení na tomto objemu. Takto utvořené body obecně nemusejí ležet v rovině, a soustava potom nebude mít řešení. Metoda nejmenších čtverců aproximuje koeficienty $\boldsymbol{a}_k$, $\boldsymbol{b}_k$ a $\boldsymbol{c}_k$ tak, aby udávaly rovnici roviny, která by měla ode všech našich bodů co možná nejmenší kvadrát vzdálenosti. Na této rovině potom nalezneme řešení v aktuálním uzlu. Podrobnější popis metody viz [3, str. 188].

Pro náš případ budeme řešit soustavu rovnic:

$$\boldsymbol{c}_k n + \boldsymbol{a}_k \sum_{j \in T(k)} x_j + \boldsymbol{b}_k \sum_{j \in T(k)} y_j = \sum_{j \in T(k)} \boldsymbol{w}_j \quad (6)$$

$$\boldsymbol{c}_k \sum_{j \in T(k)} x_j + \boldsymbol{a}_k \sum_{j \in T(k)} x_j^2 + \boldsymbol{b}_k \sum_{j \in T(k)} x_j y_j = \sum_{j \in T(k)} x_j \boldsymbol{w}_j \quad (7)$$

$$\boldsymbol{c}_k \sum_{j \in T(k)} y_j + \boldsymbol{a}_k \sum_{j \in T(k)} x_j y_j + \boldsymbol{b}_k \sum_{j \in T(k)} y_j^2 = \sum_{j \in T(k)} y_j \boldsymbol{w}_j, \quad (8)$$

kde n je počet konečných objemů, do kterých aktuální uzel náleží. x_j a y_j značejí pozici středu příslušného konečného objemu D_j. Koeficienty $\boldsymbol{a}_k$, $\boldsymbol{b}_k$ a $\boldsymbol{c}_k$ jsou vektory, neboť pro každou složku řešení máme různou hodnotu $\sum w_j$. Sčítáme zde opět hodnoty všech sousedů.

Soustavu napíšeme v maticovém tvaru:

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b},$$

zkráceně $[\mathbf{A}|\boldsymbol{b}]$, kde $\mathbf{A}$ bude matice soustavy, $\boldsymbol{x}$ vektor řešení a $\boldsymbol{b}$ vektor pravých stran. Je tedy:

$$\mathbf{A} = \begin{pmatrix} n & \sum x & \sum y \\ \sum x & \sum x^2 & \sum xy \\ \sum y & \sum xy & \sum y^2 \end{pmatrix}$$

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{c} \\ \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix} \qquad \boldsymbol{b} = \begin{pmatrix} \sum \boldsymbol{w} \\ \sum x\boldsymbol{w} \\ \sum y\boldsymbol{w} \end{pmatrix}$$

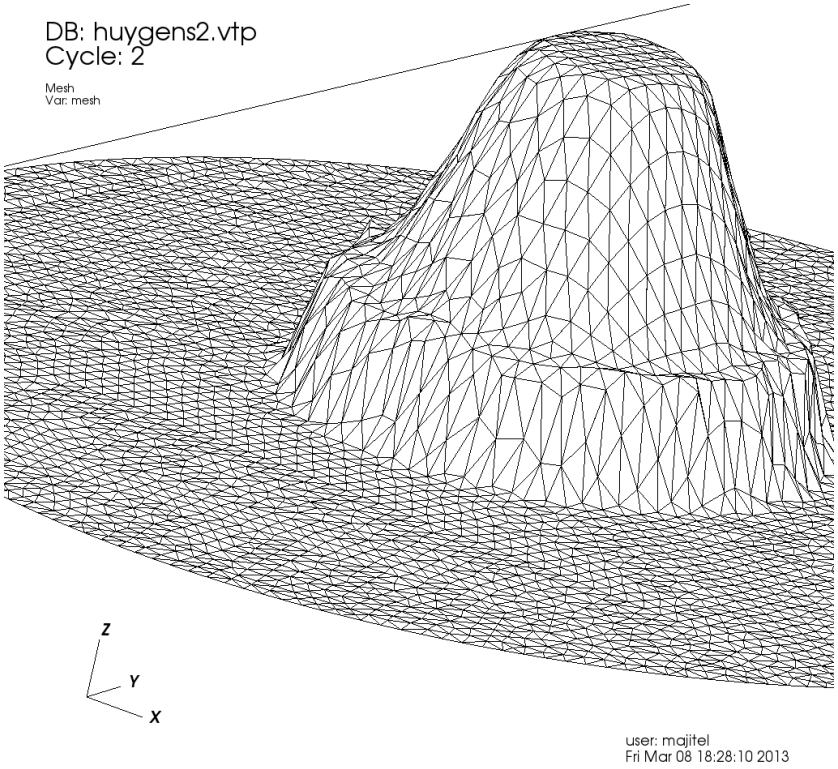
Pokud je soustava $[\mathbf{A}|\boldsymbol{b}]$ regulární, existuje právě jedno řešení $\boldsymbol{x}$. Vypočteme jej pomocí Cramerova pravidla jako

$$\boldsymbol{x}_i = \frac{det\mathbf{A}_i}{det\mathbf{A}},$$

kde $\mathbf{A}_i$ je matice $\mathbf{A}$, jejíž i-tý sloupec je nahrazen vektorem pravých stran $\boldsymbol{b}$. Případ, kdy je matice singulární, a tedy $det\mathbf{A} = 0$, je ošetřen v programu. Podrobněji viz [3, str. 121].

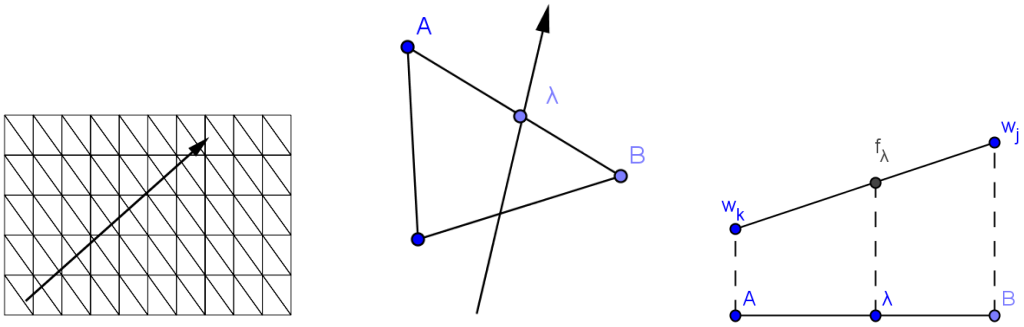
Transformaci dat z konečných objemů na uzly budeme nazývat *lineární rekonstrukcí* řešení $\boldsymbol{w}$. Na trojúhelníkové síti totiž lze tři sousední řešení na uzlech proložit rovinou. Ve čtyřúhelníkové síti tomu tak obecně není, čtyři body v prostoru nemusejí ležet v jedné rovině. Termín lineární rekonstrukce budeme v dalším textu užívat i přesto. Jelikož řešením je vektor hodnot,

můžeme jej rekonstruovat celý nebo jen jeho konkrétní složku. Při vizualizaci pak jednu jeho složku volíme jako třetí souřadnici a dostaneme nad naší dvojrozměrnou sítí trojrozměrný model řešení (viz obrázek 6).



Obrázek 6: Síť po lineární rekonstrukci řešení

Poté, co máme model lineárně rekonstruován, můžeme určit řez, podél kterého data ze sítě extrahovat. Data na jeho průsečících s hranami sítě se pak vypočítají z dat na krajních bodech hrany podle stejného poměru, v jakém řez hranu dělí (analyticky viz obrázek 7 a rovnice 9).



Obrázek 7: Řez po lineární rekonstrukci

$$\lambda \in < 0, 1 >$$
$$f(0) = w_k$$
$$f(1) = w_j$$

$$f(\lambda) = w_k(1 - \lambda) + w_j\lambda, \tag{9}$$

kde w_k, resp. w_j je jedna složka řešení na uzlu k, resp. j. Lineární rekonstrukce však není jedinou možností, jak transformovat data. Můžeme jen použít určitou funkci, kterou budeme uplatňovat na všechny konečné objemy sítě. Vektor řešení $\boldsymbol{w}$ přitom bude jejím parametrem. Můžeme také chtít použít celý vektor funkcí, jednu funkci pro jednu jeho složku. Extrakcí dat ze sítě konečných objemů rozumíme buď lineární rekonstrukci nebo transformovaná data. V tabulce 1 uvádíme přehled, jakou podobu bude mít transformovaný vektor řešení $\boldsymbol{w}$.

	pomocí funkce	pomocí vektoru funkcí
na konečné objemy	skalár	vektor
na uzly	skalár	vektor

Tabulka 1: Transformace dat

2.3 Formáty pro vizualizaci

2.3.1 VTK

VTK (*Visualisation Toolkit*) od firmy Kitware je systém pro vizualizaci vědeckých výpočtů. Grafické uživatelské rozhraní neobsahuje. Je knihovnou o stovkách tříd, nad kterými uživatel píše aplikace v programovacím jazyce C++. Dostupné jsou i wrappery pro další jazyky. VTK definuje různé formáty datových sad, a to: strukturované body, strukturovanou a nestrukturovanou mřížku a polygonální data. Právě u těch budeme zkoumat, jak se ukládají do souborů (podrobněji viz [11, str. 12]):

- Textová verze - zastaralá, neumožňuje náhodný, paralelní přístup a obsahuje pouze omezenou podporu komprese dat (příklad viz příloha C.3.1).

  Síť o 20 000 uzlech bude v textovém formátu zabírat přibližně 3,5 MB (viz tabulka 3).

- XML verze je v současné době standardem (příklad viz příloha C.3.2). Formát byl navrhnut, aby bylo možné pracovat s daty také paralelně, načítat a ukládat je v aplikacích, které pracují i ve více procesech. Tuto možnost však v našem programu nevyužíváme.

  Tatáž síť, jako v předchozím případě, bude v tomto XML formátu zabírat přibližně 8 MB (viz tabulka 3) a analogicky tomu se bude ve vizualizačních systémech i poměrně dlouho načítat.

Tento formát umožňuje uložit do souboru celý vektor řešení, avšak pouze v jednom konkrétním čase. Pokud tedy budeme chtít vytvářet animace a zobrazit průběh výpočtu, musíme ukládat stavy sítě z jednotlivých iterací do samostatných VTK souborů. Takovou animaci nyní již zmíněné programy umožňují zobrazit. Úvod do VTK najde čtenář také v češtině v [20, str. 25].

2.3.2 Další formáty

Z tabulky 2 je zřejmé, že nejlepší podporu software má formát VTK, který je standardem. Proto jsme se rozhodli využít jej i v našem programu a další formáty pouze zmíníme. V tabulce 3 potom porovnáváme jejich velikosti. Nejúspornější je formát Exodus.

-	VTK	Exodus	VRML	PLOT3D	XDMF
Mayavi2	+	1/2[1]	+	-	-
ParaView	+	+	+	-	+
VisIt	+	+	-	-	+
gnuplot	-	-	-	-	-
MVE-2	-	-	1/2[2]	-	-

Tabulka 2: Formáty souborů - podpora ve vizualizačním software

Exodus Binární formát, který podporuje ukládání několika sítí v jednom souboru (blíže viz [12, str. 263]).

VRML XML soubor pro zobrazení 3D dat. Uvedené programy jej načítají přes knihovnu VTK, kvůli omezení implementace však není povoleno texturování (zdroj: [15]).

PLOT3D Formát je rozdělen na dva soubory. Topologie sítě má příponu .xyz a data nad ní příponu .q. Formát se nám nepodařilo v žádném programu načíst (viz 3.7). Exportovat jej dokázal jen ParaView.

[1] Mayavi2 nepodporuje u formátu Exodus časové série. Načte jen první snímek.
[2] MVE-2 nenačte ze souboru VRML data.

XDMF Taktéž formát, rozdělený do dvou souborů. Metadata jsou uložena v XML formátu s příponou .xmf. Ten odkazuje na definované pozice do binárního souboru HDF5, ve kterém je uložena topologie i data. Programy však u tohoto formátu nepodporují animace ani pomocí sérií souborů.

formát	číslo iterace	velikost [B]
Exodus	1 + 2 + 3	6 033 804
VTK - ASCII	1	3 403 109
VTK - ASCII	2	3 403 109
VTK - ASCII	3	3 403 109
VTK - XML	1	5 748 056
VTK - XML	2	8 321 749
VTK - XML	3	8 651 480
XMDF - data	2	3 258 532

Tabulka 3: Formáty souborů - porovnání velikostí

2.3.3 Použitý formát

V našem programu používáme formát VTK, neboť umožňuje uložit nad sítí i více složek dat a jelikož představuje standard. S jeho podrobným popisem se lze setkat i v dokumentaci programů, které jsme prozkoumali. Použijeme jeho aktuální XML verzi s textovými hodnotami dat, neboť jejich export může být proveden přímým kopírováním z paměti. Pro další vývoj programu navrhujeme převést tyto hodnoty do binární či komprimované podoby (viz 3.8).

2.4 Programy pro vizualizaci

Z programů, které se používají pro vizualizaci vědeckých dat, jsme vybrali opensource programy Mayavi2, VisIt, ParaView, Gnuplot a lokální MVE-2[3]. Pro různé způsoby vizualizace obsahují sadu modulů a filtrů, kterými je možné data ještě před zobrazením přetransformovat. Pro každý z programů uvedeme seznam načítaných a ukládaných formátů, avšak omezíme se přitom na formáty obecně známé a na ty, které jsme popsali v kapitole 2.3. Pro další formáty odkážeme na příslušnou dokumentaci. Zmiňme ještě alespoň, že programy jsou modulární a pro jakýkoliv další formát si do nich lze poměrně snadno připsat vlastní plugin. Spuštění programů trvá 20-30 s, což je vzhledem k jejich složitosti nasnadě.

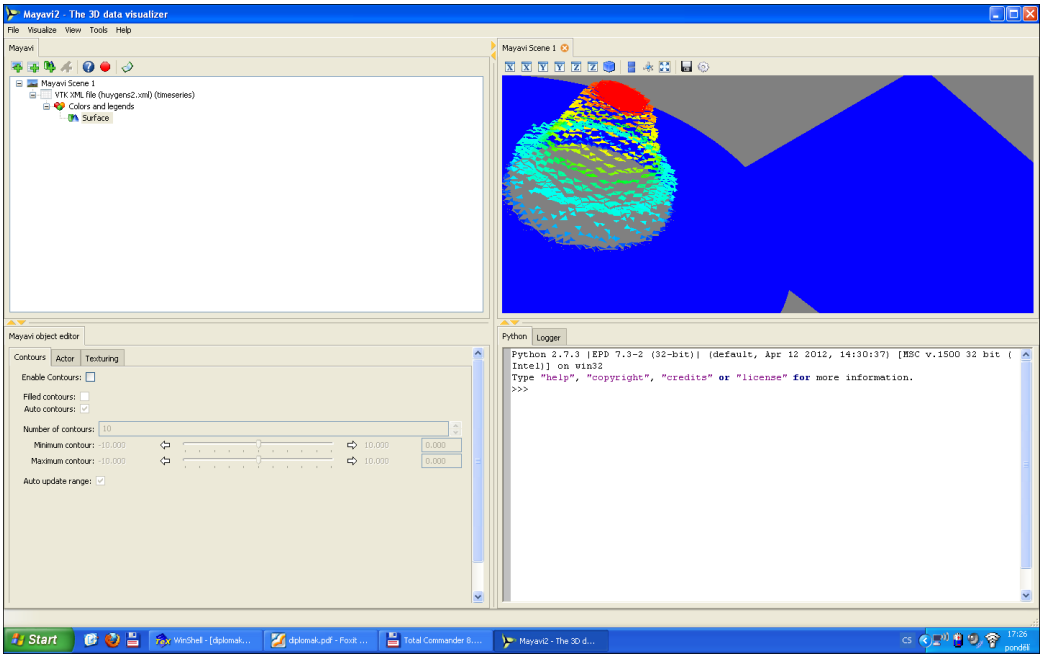
[3]Všechny projekty jsou aktivní a jejich vývoj ke květnu 2013 pokračuje.

2.4.1 Mayavi2

Program Mayavi2 je pevnou součástí distribuce jazyka Python Enthought EPD. Ta je komerční, akademickou licenci však nabízí zdarma. Program využívá API dalších technologií v této distribuci, proto není šířen samostatně, jako jeho předchůdce Mayavi1. Program je sám napsán v jazyce Python a pro vizualizaci využívá knihovnu VTK (viz 2.3.1). Kromě GUI aplikace přichází i s vlastním skriptovacím jazykem a interpretrem, podobným jako v programu Gnuplot (viz 2.4.4). Jako standardy načítání dat jsou na titulní stránce dokumentace ([14]) uvedeny VTK soubory a poté datové sady, přímo generované dalšími pluginy Pythonu TVTK (Wrapper pro VTK na bázi rozšíření Pythonu s názvem Traits), mlab a NumPy. Tuto druhou možnost jsme neprozkoumali (viz 3.7). Na obr. 8 představujeme GUI programu. Další formáty dat jsou zmíněny v [15].

Načítání: VTK rodina (.vtk, .xml, .vtp, ...), obrázky (.png, .jpg, .bpm, .tiff), Plot3D (.xyz, .q), VRML2 (.wrl), Waveform OBJ (.obj), Exodus (.exii), 3D Studio (.3ds), DICOM (.dcm)

Ukládání: obrázky (.png, .jpg, .bpm, .tiff), VRML2 (.wrl), Waveform OBJ (.obj), Renderman (.rib)



Obrázek 8: Mayavi2

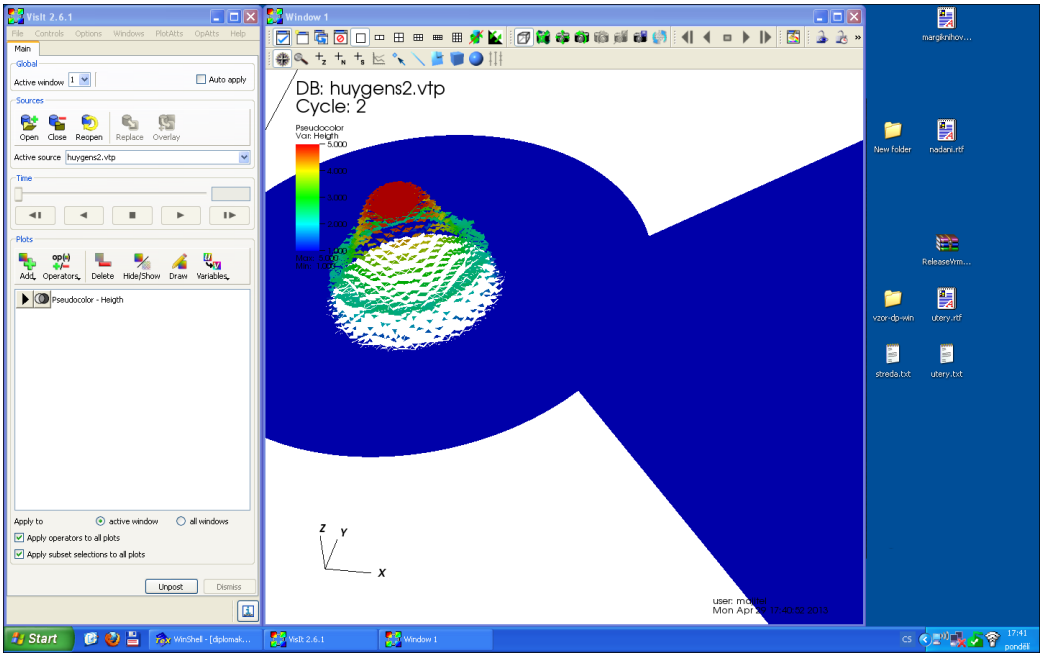
2.4.2 VisIt

VisIt byl původně vyvinut pro zobrazení velkých geodetických dat. S časem nabyl funkcionality i mimo tento obor do oblasti běžných simulací a autoři jej uvolnili pro veřejnost. Program je multiplatformní opensource pod BSD licencí. Jeho architektura je distribuovaná, a umožňuje tak paralelní výpočty a zobrazení dat v místě, kde byla generována, bez potřeby jejich přenášení. Kromě GUI může být program ovládán z programovacích jazyků C++, Java a Python. Je snadno rozšiřitelný prostřednictvím pluginů.

Pro načítání souborů opět preferuje VTK soubory. Přichází také s vlastním formátem SILO a knihovnou v jazycích C a Fortran, která do něj převádí (viz [10, str. 13]). Program nepodporuje formát VRML. Kompletní seznam podporovaných formátů nalezne čtenář v [9, str. 24]. Na obr. 9 představujeme GUI programu, rozdělené do více oken.

Načítání: VTK rodina (.vtk, .xml, .vtp, ...), SILO (.silo), obrázky (.png, .jpg, .bpm, .tiff), Exodus (.e, .exii, ...), Plot3D (.xyz, .q), TecPlot (.tec), Waveform OBJ (.obj), XDMF (.xmf), XMDV (.ocx)

Ukládání: obrázky (.png, .jpg, .bpm, .tiff), VTK rodina (.vtk, .xml, .vtp, ...), SILO (.silo), TecPlot (.tec), XMDV (.ocx)



Obrázek 9: VisIt

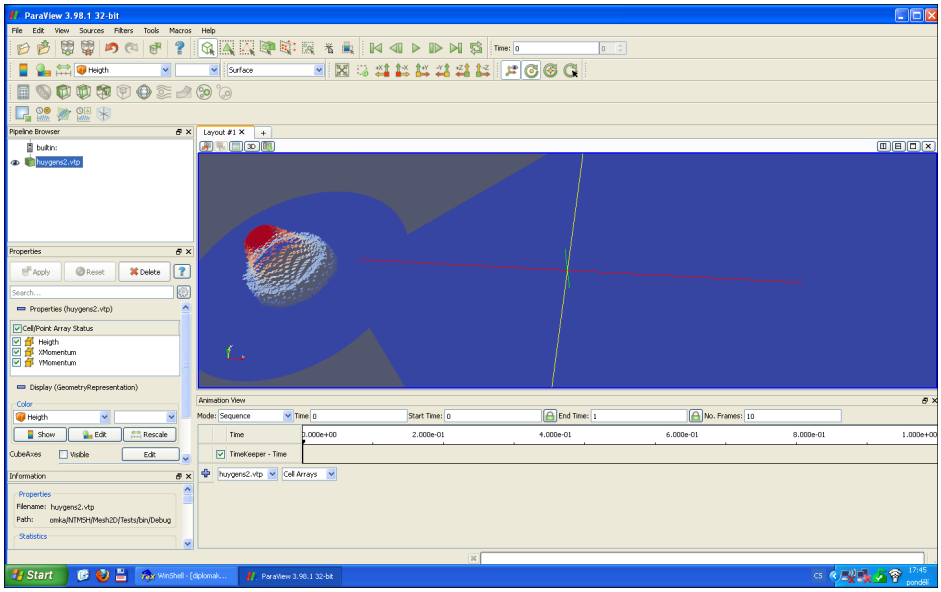
2.4.3 ParaView

ParaView vyrobila firma Kitware, od níž pochází také formát VTK. Ten je zde proto standardem. Program je multiplatformní opensource a je připraven na velká data pro superpočítače či distribuované systémy. Program umožňuje skriptování prostřednictvím jazyka Python. Umožňuje exportovat formát VRML ([12, str. 118]) a také formát Exodus (viz 2.3.2), čímž se liší od výše uvedených programů. Detailní seznam podporovaných formátů je uveden v [12, str. 260].

Na obr. 10 představujeme GUI programu, které je narozdíl od VisIt jsou všechny moduly programu v jediném okně. Další pluginy lze napsat jako třídy jazyka C++. Na Katedře informatiky a výpočetní techniky ZČU bylo vyvinuto rozšíření tohoto programu o stereoskopický renderer, který zpřístupňuje v programu práci na dvou monitorech, umožňuje pro scénu nastavit vzálenost od oka a rozdělí ji na stereostěně na levý a pravý kanál ([13, str. 5]).

Načítání: VTK rodina (.vtk, .xml, .vtp, ...), obrázky (.png, .jpg, .bpm, .tiff), Exodus (.e, .exii, ...), Plot3D (.xyz, .q) VRML2 (.wrl), Waveform OBJ (.obj), XDMF (.xmf)

Ukládání: VTK rodina (.vtk, .xml, .vtp, ...), obrázky (.png, .jpg, .bpm, .tiff), Exodus (.e, .exii, ...), XDMF (.xmf), PDF (.pdf), VRML (.wrl, .vrml)



Obrázek 10: ParaView

2.4.4 Gnuplot

Gnuplot je multiplatformní, vyvíjený komunitou. Není však opensource, ale přichází s vlastní licencí, která zaručuje její bezplatnost. Narozdíl oproti výše uvedeným programům se ovládá z příkazové řádky a přichází s vlastním skriptovacím jazykem (příklad viz kód 1 a obr. 11, popis viz [16]). Data načítá z prostých textových souborů, kde jsou uspořádána do sloupců, oddělených tabulátorem. V našem programu generujeme takové soubory z vypočtené sítě nebo po provedení řezu (viz 2.5.6). Jejich nevýhodou je, že si uživatel musí pamatovat, co který sloupec představuje. V programu Gnuplot pak data vykreslí příkaz

`plot 'C:\tmp\leftright-height-flat.txt' using 1:5`

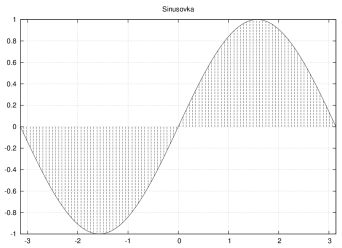
Tím požadujeme vykreslit dvojrozměrný graf s x-ovou souřadnicí z prvního sloupce a y-ovou z pátého. Tento datový soubor a vygenerovaný obrázek 20 nalezne čtenář v příloze C.4.

Načítání: pouze vlastní datový formát

Ukládání: obrázky (.png, .jpg, .bpm, .tiff), PDF (.pdf), PostScript (.ps, .eps), SVG (.svg)

Kód 1: Příklad skriptu

```
1 set terminal postscript
2 set output "sinusovka.ps"
3 set grid
4 set nokey
5 set title "Sinusovka"
6
7 plot [-3.14:3.14] sin(x), sin(x) with impulses
```

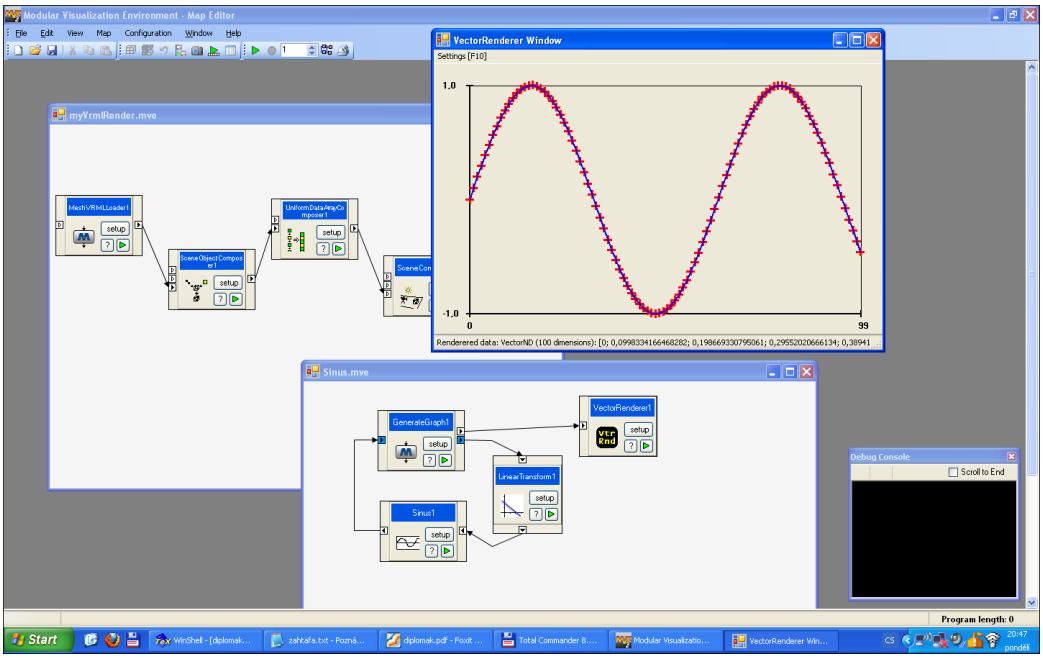


Obrázek 11: Funkce sinus, generovaná skriptem pro program Gnuplot

2.4.5 MVE-2

MVE-2 je modulární prostředí pro operace s grafickými daty, vyvíjený přímo na naší Katedře informatiky Fakulty aplikovaných věd ZČU. Primárním případem užití programu je vizualizace dat, které program přímo vygeneruje. Ten je nicméně relevantní i pro tuto práci, neboť jeden jeho modul umožňuje načítání trojúhelníkové sítě, uložené do formátu VRML. Ač tento formát umožňuje ukládání dat na síti, modul programu MVE-2 je nenačítá a lze tak pracovat jen se sítí samotnou. Nám se však nepodařilo ani to a program skončil při pokusu o vykreslení chybou (viz 3.7). Uživatel vytváří algoritmus v podobě grafu. Síť může prostřednictvím dalších modulů transformovat nebo např. přidávat světla.

Prostředí je díky modularitě snadno rozšiřitelné (viz [19, str. 42]). Již jsou napsány moduly pro vizualizaci nestrukturované sítě pomocí různých technologií - např. softwarový SceneRenderer (viz [20, str. 37]) nebo renderer pro DirectX 10, akcelerovaný na GPU. Na obr. 12 představujeme GUI a graf modulů, který vygeneruje funkci sinus.



Obrázek 12: MVE-2

2.4.6 Použití animací

Součástí GUI uvedených programů jsou ovládací tlačítka pro pohyb v čase. Gnuplot, který GUI nemá, pouze přehraje předem naskriptované animace, obdobně jako program MVE-2. V ostatních programech můžeme jednotlivými snímky animace procházet. Každý takový snímek bude samostatným modelem. Jeden VTK soubor může obsahovat jen jeden takový model v jednom čase, a časová série se tedy načítá pomocí série VTK souborů. Ty jsou rozpoznány při vhodném pojmenování souborů (fooN.vtk, případně foo-N.vtk či foo_N.vtk, kde foo je libovolný název a N je celé číslo). Program VisIt navíc vyžaduje ve stejném adresáři speciální soubor se seznamem souborů v časové sérii a direktivou *!NBLOCKS*, která udává počet souborů, sdružených do jednoho časového kroku. Tento soubor má příponu .visit a jeho formát uvádíme v kódu 2 (viz [9, str. 31]).

Kód 2: Schéma souboru .visit

```
1 !NBLOCKS 1
2 combo-0.vtp
3 combo-1.vtp
4 combo-2.vtp
5 ...
```

Animaci z časových sérií lze přehrát v programech VisIt a ParaView. Program Mayavi2 má v tomto směru omezenou funkcionalitu a ze série v něm lze zobrazit pokaždé jen jednu konkrétní síť po posunu jezdcem. Omezením přehrávání takových animací je, že program musí znovu generovat celý model pokaždé, co změníme snímek. To trvá přibližně 3 s a při přehrávání tak nastávají dlouhé pauzy, kdy je CPU plně vytíženo výpočtem.

Program VisIt umožňuje vygenerovat video do formátu MPEG (viz CD). Program ParaView potom do kontejnerů AVI či OGG Theora. Oba programy také podporují export obrázkových sérií.

2.5 Původní stav programu

K výpočtu Eulerových rovnic (2) byl vytvořen samostatný program NTMSH s konzolovým rozhraním, který používá metodu konečných objemů na síti o dvou rozměrech. Kromě výpočtu samého je jeho cílem poskytnout uživateli možnost s výslednými daty pracovat, transformovat je, extrahovat je ze sítě podél určeného řezu nebo je exportovat do souboru, prostřednictvím kterého je bude možno ve vhodném software vizualizovat. Program byl napsán v jazyce C# a pro jeho kompilaci bylo použito prostředí Mono (viz [17]). Objektová struktura programu se nachází v příloze B.1. Pro všechny jeho funkčnosti byly napsány testy.

2.5.1 Načítání sítě

Síť načítáme do objektu *FVMMesh* (Finite Volume Method Mesh) ze souborů prostřednictvím tříd, implementujících rozhraní *IMeshLoader* (viz příloha B.1). Oba dosud implementované loadery načítají soubory s touž příponou MSH. Formát těchto souborů je však dvojí:

- *FFMshLoader* načítá síť, kterou vygeneroval program *FreeFem++* a která je vždy trojúhelníková. Příklad nalezne čtenář v příloze C.1. Pro podrobné schéma souboru odkážeme na [4, str. 94].
- *GMshLoader* načítá formát softwaru *GMsh*. Počítá kromě trojúhelníkové také s čtyřúhelníkovou sítí, nebo sítí, kde se oba geomerické útvary kombinují. Program navíc vygeneruje vždy 3D síť, při načítání proto v našem případě nebudeme brát zřetel na třetí souřadnici. Příklad souboru je v příloze C.2. Podrobněji v [5, kap. 9.1].

Tímto postupem tedy načteme do objektu *FVMMesh* topologii sítě, budou v něm však chybět data. Nainicializovat je před samotným výpočtem musí ještě uživatel sám.

2.5.2 Inicializace dat

Objekt *FVMMesh* se skládá z konečných objemů, které reprezentují objekty *Volume*. Řešení $\boldsymbol{w}_i$ ukládáme v tomto objektu do veřejného pole `public double W [][]`, kde první složkou je časový okamžik a druhou komponenta vektoru. *SolutionInitializer* nastavuje objektům *Volume* počáteční řešení $\boldsymbol{w}_i^0$ (viz rovnice 4) do jednotlivých složek pole W[0][]. K výpočtu integrálu samého používáme Gaussovu kvadraturu, podrobněji viz [7, str. 152pp].

2.5.3 Konfigurace výpočtu

Po inicializaci dat lze tedy přejít k výpočtu. Jeho parametry nastaví uživatel v konfiguračním XML souboru (podrobněji viz příloha D):

- *Použitý numerický tok*, např. Vijaysundaramův, Stegerův-Warmingův (viz 2.2.2).
- *Maximální hodnota časového kroku*, jak dlouho trvá jedna iterace v metodě konečných objemů.
- *CFL*$\in (0, 1)$ je číslo, omezující časový krok z důvodu numerické stability výpočtu (blíže viz [8]).
- *Zastavovací podmínka*, tj. buď maximální iterace, maximální čas nebo rozdíl normy řešení posledních dvou iterací.
- *Pojmenování složek řešení* pro popis os na výsledném grafu.

Příklad konfigurace uvádíme v příloze D.

2.5.4 Použití

Použití programu si představíme na jednom hotovém testu[4]. Nejprve načteme síť konečných objemů ze souboru na adrese *path* :

Kód 3: Použití programu z pohledu uživatele

```
1 FFMshLoader loader = new FFMshLoader(File.OpenText(path))↩
      ;
2 Mesh2D.FVMMesh mesh = new Mesh2D.FVMMesh(loader);
```

Potom u ní nastavíme okrajové podmínky (viz rovnice 2). K vytvoření hranice používáme konstruktor
`Boundary{BoundaryDelegate fce, BoundaryType type, double[] boundary_data↩ , int id, double tmin, double tmax)`,
kde *fce* je delegát funkce pro popis jednoho okraje, který vychází z parametru matematické funkce t. Jeho minimální a maximální hodnotu pak označují parametry *tmin* a *tmax*. V našem případě probíhá parametr t od -1 do 1 ve všech složkách a celkovou hranicí je tak čtverec. Po implementaci pohyblivé sítě bude sloužit k tomu, aby se zabránilo v posunu uzlů mimo ohraničenou oblast.

[4] Test i všechny kódy v této kapitole jsou z modulu *SweHuygens.cs*

```
3  const double hin=5;
4  const double hout=1;
5  ...
6  double [] wall=new double [] {hout,0,0};
7  mesh.AddBoundary(new Boundary((double t)=>new Vector2D(1,↩
       t),BoundaryType.Fixed,wall,1,-1,1));
8  mesh.AddBoundary(new Boundary((double t)=>new Vector2D(t↩
       ,1),BoundaryType.Fixed,wall,2,-1,1));
9  ...
```

U všech konečných objemů sítě inicializujeme počáteční stav, tedy vektor $\boldsymbol{w}^0$. V tomto případě jde o problém mělké vody (viz 2.1.3), proto bude mít tento vektor za složky výšku hladiny a momenty proudění ve směrech os. Nyní nastavíme počáteční podmínky. Potřebujeme k tomu funkce, přes které budeme integrovat. Pro příklad si uvedeme:

Kód 4: Funkce pro inicializaci výšky hladiny, jedné složky řešení

```
static double InitHeight(double x,double y)
{
  if(Math.Sqrt(x*x+y*y)<=2)
    return hin;
  else
    return hout;
}
```

Takové funkce přidáme do společného objektu *InitialCondition* a inicializujeme pomocí něj počáteční podmínky:

```
8  InitialCondition ic=new InitialCondition(configuration);
9  ic.AddInitialCondition(0,InitHeight);
10 ic.AddInitialCondition(1,InitXMomentum);
11 ic.AddInitialCondition(2,InitYMomentum);
12 SolutionInitializer.Initialize(mesh,ic,configuration);
```

Metoda *AddInitialCondition* má dva parametry. Prvním je pořadové číslo funkce a druhým funkce sama. Každou z nich pak *SolutionInitializer* v metodě *Initialize* integruje přes každý konečný objem zvlášť a výsledek mu nastavuje jako počáteční podmínku (viz 2.5.2).

Nyní již můžeme inicializovat samotný výpočet (objekt *EFVMSolver*), nastavit mu obsluhy událostí a spustit jej:

```
15 EFVMSolver solver= new EFVMSolverSweLCB(mesh,↩
       configuration);
16 solver.AfterComputationalStep+=AfterComputationalStep;
17 solver.OnEnd+=OnEndComputation;
18 solver.Solve();
```

Pro ilustraci uvedeme i jednu obsluhu události, kterou definujeme ve stejném modulu. Půjde o událost *AfterComputationalStep*, která se vyvolá po skončení jednoho kroku výpočtu, tedy poté, co budou ve všech konečných objemech sítě přepočtena řešení. Obsluhy mají za parametr objekt třídy *EFVMContext*, která především obaluje objekty *EFVMSolver* a *FVMMesh* spolu s aktuálním číslem iterace a časem ve výpočtu, kdy byla událost vyvolána. Tento objekt je vytvářen objektem *EFVMSolver* pokaždé znova, když je vyvolána událost, a je přitom inicializován aktuálními hodnotami. Uživatel pak přes tento objekt získá přístup k aktuálnímu stavu výpočtu a k síti.

Kód 5: Obsluha události AfterComputationalStep

```
1 static void AfterComputationalStep(EFVMSolverContext ↩
      context)
2 {
3   Console.WriteLine ("Iteration {1} actual time {0:G4} ↩
       with timestep={2}.", context.PhysicalTime, ↩
       context.Iteration, context.TimeStep);
4 }
```

Po výpočtu budeme na každém konečném objemu D_i znát celý vektor řešení $\boldsymbol{w_i}$. Ten bude uložen v objektové hierarchii objektu *FVMMesh*. Nyní rozebereme podobněji jednotlivé důležité části programu.

2.5.5 Detailní rozbor průběhu výpočtu

Výpočet probíhá v objektu *EFVMSolver*, jehož součástí je objekt *EFVMFlux* pro výpočet numerického toku $\boldsymbol{H}$ z rovnice 2.

1. Iterační smyčka

   Nyní podrobněji probereme iterační smyčku v metodě *Solve()*. Může trvat velmi dlouho, než se zastaví, nezřídka celé dny. Aktuální stav proto může uživatel sledovat pomocí událostí, které jsou ve smyčce vyvolávány. Jejich parametrem je přitom pokaždé objekt *EFVMContext*. Uživatel naprogramuje obsluhu události a přes tento objekt přistupuje k aktuálnímu stavu výpočtu. První událostí je *OnStart*, která se vyvolá hned po inicializaci proměnných:

Kód 6: Metoda Solve()

```
1  double timestep=0; int iteration = 0; double time = ↩
       0;
2  OnStart(new EFVMSolverContext(this,time,timestep,↩
       iteration));
```

Poté metoda iteruje ve smyčce dokud nenastane zastavovací podmínka, definovaná v konfiguračním souboru. Jako první se určí časový krok τ. Hodnotu τ (viz rovnice 5) počítá objekt *EFVMFlux* a ovlivňuje jej CFL podmínka (viz popis konfigurace, kapitola 2.5.3).

```
3  while(time<_config.StopTime && iteration<_config.↩
       MaxIteration && _running)
4  {
5    timestep=Math.Min(_flux.Tau,_config.TimeStep);
```

Poté se přistoupí k samotnému výpočtu v metodě *Step(double timestep)*. Před ní i po ní se vyvolají příslušné události. Navíc se vypočítá aktuální norma pro zastavovací podmínku stacionárního řešení.

```
6    BeforeComputionalStep(new EFVMSolverContext(this,↩
       time,timestep,iteration));
7    Step(timestep);
8    actualsteadystatenorm = ...
9    AfterComputationalStep(new EFVMSolverContext(this,↩
       time,timestep,iteration,actualsteadystatenorm))↩
       ;
```

Pokud zastavovací podmínka nenastala, staré řešení přepíše novým a pokračuje se dalším iteračním krokem. Pokud ano, vypočítali jsme pro každý konečný objem D_i výsledné řešení $\boldsymbol{w}_i$. Na konci se jen vyvolá událost *OnEnd*.

```
10    if(actualsteadystatenorm <_config.SteadyStateNorm)
11      break;
12    foreach (Volume voli in _mesh.Volumes) {
13      for (int i = 0; i < _dimension; i++) {
14        voli.W[0][i] = voli.W[1][i];
15      }
16    }
17    time=time+timestep; iteration++;
18   }
19   OnEnd(new EFVMSolverContext(this,time,timestep,iteration↩
       ));
```

2. Krok výpočtu

   V metodě *Step(double timestep)* vypočteme aktuální řešení, vektor $\boldsymbol{w}_i^{k+1}$ na všech konečných objemech sítě. Iterujeme tedy přes ně a každému konečnému objemu počítáme součet toků $\sum \boldsymbol{H}|\Gamma_{ij}|$ od jeho sousedů (viz rovnice 5). Nejprve inicializujeme globální pole _sum_H[]:

Kód 7: Metoda Step(double timestep)

```
1  double [] results; double lambda;
2  foreach(Volume voli in _mesh.Volumes)
3  {
4    for(int dim=0;dim<_dimension;dim++)
5    {
6      _sum_H[dim]=0;
7    }
```

Jeho jednotlivé složky budou představovat tok příslušné fyzikální veličiny aktuálním konečným objemem D_i. Vypočítáme jej jako součet toků od všech jeho sousedů.

```
8     for(int j=0;j<voli.Neighbours.Length;j++)
9     {
10      results=_flux.H(voli,j);
```

Vektor *results* představuje tok veličin z konečného objemu D_j do aktuálního D_i. Pro každou komponentu jej pak přenásobíme délkou hranice $|\Gamma_{ij}|$ (viz rovnice 5) a výsledek přičteme k součtové proměnné.

```
11      for(int dim=0;dim<_dimension;dim++)
12      {
13        _sum_H[dim]+=(results[dim]*voli.Edges[j].Length);
14      }
15    }
```

V tento moment máme vypočítán součet toků od všech sousedů aktuálního konečného objemu. Tento celkový tok je nyní nutno násobit členem $\frac{\tau}{|D_i|}$ (viz rovnice 5). Novou hodnotu řešení $\boldsymbol{w}_i^{k+1}$ ukládáme do pole W[1] aktuálního konečného objemu.

```
16    lambda=timestep/voli.Area;
17    for(int dim=0;dim<_dimension;dim++)
18    {
19      voli.W[1][dim]=voli.W[0][dim]-lambda*_sum_H[dim];↩
          }
20  }
```

Poznamenejme ještě na tomto místě znovu, že takto vypočítané nové řešení voli.W[1] bude překopírováno do starého voli.W[0]. Děje se tak v metodě *Solve()* (viz kód 6, řádek 14).

2.5.6 Transformace a extrakce dat

Na síti *FVMMesh* s vypočtenými daty můžeme provádět pomocí třídy *DataSlicer* lineární řezy podél specifikované úsečky (viz obrázek 7 na str. 15).

Kód 8: Příklad extrakce dat podél daného řezu

```
1 Vector2D start=new Vector2D(-5,0);
2 Vector2D end=new Vector2D(5,0);
3 GPlotDataSlicer gds= new GPlotDataSlicer(mesh,↩
     configuration,start,end);
```

Před řezem ovšem musíme transformovat data z konečných objemů na uzly. Použijeme k tomu metodu nejmenších čtverců (viz kapitola 2.2.3), objekt třídy *LSSolutionTransformation* (Least Square). Ten obsahuje metodu *ComponentLReconstruction*, pomocí níž provedeme lineární rekonstrukci (v uvedeném příkladě parametr 0 znamená rekonstrukci jen na základě první složky vektoru řešení $\boldsymbol{w}$). Postup pro uložení řezu tedy bude vypadat následovně:

```
4 LSSolutionTransformation lsstrans=new ↩
     LSSolutionTransformation(mesh,configuration);
5 gds.SaveComponentSlice("lssdensity.txt",lsstrans.↩
     ComponentLReconstruction(0));
```

Do souboru ukládáme postupně hodnoty první komponenty výsledného řešení na průsečících dané úsečky s hranami sítě.

2.5.7 Vizualizace

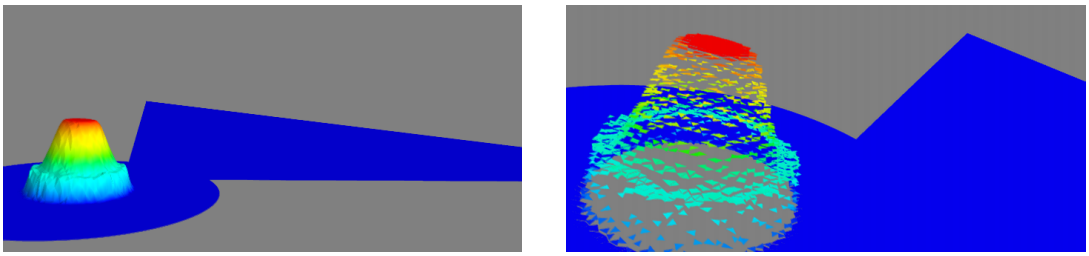
Objekt *FVMMesh* s vypočtenými daty vizualizujeme prostřednictvím formátu VTK (viz 2.3.1). Zůstaneme u příkladu z modulu *SweHuygens.cs* a opět si nejdříve představíme, jak by to to uživatel provedl:

Kód 9: Export sítě s vypočtenými daty do souboru

```
1 VTKXmlDataVisualisator visualiser=new ↩
     VTKXmlDataVisualisator(File.CreateText("huygens2.xml")↩
     ,mesh,configuration);
2 visualiser.VisualiseSolution(SolutionType.Points,0);
3 visualiser.Write();
```

Metoda *VisualiseSolution* má za první parametr typ vizualizovaných dat, druhým složku řešení. V našem případě (*SolutionType.Points*) zobrazujeme řešení na uzlech. Druhou možností je zobrazit řešení po částech konstantní

(*SolutionType.Cells*). Rozdíl mezi oběma způsoby zobrazení představujeme na obrázku 13. Druhý parametr metody potom udává, kterou složku řešení budeme chtít vizualizovat na ose z (v našem případě složku s indexem 0, tedy výšku hladiny - viz inicializace výpočtu níže). Poznamenejme k tomu, že metoda *VisualiseSolution* je přetížená a pokud nemá druhý parametr, potom součástí VTK souboru mohou být i všechny složky řešení zároveň. V programech pro vizualizaci lze pak vybrat jakoukoli z nich a zobrazit ji na 2D modelu, např. pomocí rozdílné intenzity barvy.



Obrázek 13: Vizualizace dat - vlevo lineární rekonstrukce,
vpravo po částech konstantní řešení

2.5.8 Chybějící části

V programu, který jsme právě představili, zatím schází test sítě s proměnnou topologií. Jeho rozhraní by však mělo umožnit vypočítat takovou síť taktéž. Ač program dále obsahuje rozhraní pro obecné řezy na síti, implementován zatím byl pouze řez podél konkrétního vektoru (viz obr. 7 na str. 15) a např. řez podle křivky chybí. K jeho implementaci je program nicméně připraven. Především ale není možné průběh výpočtu zaznamenat a následně graficky vizualizovat. Program tedy bude nutno rozšířit o úložiště, které v kombinaci s grafickou vrstvou bude takovou možnost poskytovat.

3 Realizační část

V teoretické části jsme popsali současný stav programu a formát souborů, který používáme pro ukládání. Nyní se zaměříme na rozšíření programu o datové úložiště. Vlastností iteračních metod je, že výpočet probíhá na základě předchozího kroku. Metodu implementujeme na dvou proměnných, ukládá se řešení z předchozí a aktuální iterace (viz kapitola 2.5.5, bod 2, řádek 19). Dosud nebylo možno zjistit stav proměnné z předchozích kroků, jelikož proměnná byla již přepsána. Průběh výpočtu jsme nikam neukládali. Naším úkolem nyní bylo vytvořit takové úložiště, které by umožňovalo reprodukci výpočtu od jakékoliv jeho iterace. To vše při minimálních nárocích na čas a paměť.

3.1 Případy užití

1. ukládat stav sítě v aktuální iteraci
2. načíst stav sítě na základě iterace
3. načíst stav sítě na základě času
4. změnit v konkrétní iteraci parametry výpočtu a umožnit znovu spustit výpočet
5. odložit celé úložiště na disk
6. načíst úložiště předchozího výpočtu z disku

3.2 Vývoj modelu

Původně jsme předpokládali, že bude možno po každém kroku pouze hluboce kopírovat celou síť do pole. Stačilo by k tomu doprogramovat kopírovací konstruktor objektu *FVMMesh*. Uživatel by se tedy potom o ukládání kopie objektu staral sám v obsluhách událostí. Kromě té by musel ukládat také metadata sítě - číslo iterace a čas.

Problémem při kopírování celé sítě je však ukládání velkého množství nadbytečných dat. Síť tvoří objekty *Vertex*, *Edge*, *Boundary*, *Volume*, a ty bychom při hluboké kopii museli vytvářet nově. Mělká kopie by nestačila, neboť hodnoty řešení jsou jen v objektech *Volume*. Kopírujeme tak celou objektovou strukturu a úměrně tomu pak rostou paměťové a časové nároky. Abychom disponovali konkrétními údaji, kopírovací konstruktor *FVMMesh* jsme implementovali.

Možností, jak ušetřit paměť, by bylo ukládat jen hodnoty řešení $\boldsymbol{w}_i$. Ty jsou však uloženy poměrně hluboko v objektové struktuře, jako dvojrozměrné pole W[0][] v objektu konečného objemu *Volume* (viz 2.5.2). Bylo by proto vhodné, poskytnout uživateli standardní metody pro kopii těchto hodnot ze všech konečných objemů sítě. Uložit kopii celého objektu *Volume* není výhodné, neboť ukládá spolu s řešením zároveň část informace o topologii - *Volume* má odkazy na objekty uzlů a hran, ze kterých se konečný objem skládá. My budeme tedy potřebovat kopírovat pole reálných čísel W[0][] do jiné datové struktury. Dobrým řešením je dvojrozměrné pole, kde prvním rozměrem je index konečného objemu a druhým komponenta řešení. Porovnání tohoto řešení s kopírováním celé sítě uvádíme v tabulce 4[5].

	čas [s]	paměť [MB]
hluboká kopie *FVMMesh*	0,8070	25,3357
pouze řešení $\boldsymbol{w}_i$	0,0441	2,3634

Tabulka 4: Nároky na ukládání jedné sítě

Tento model předpokládá, že se objekt sítě nezmění. Ve většině případů tomu tak skutečně bude. Zadání jsme však měli takové, že se síť během výpočtu měnit může a už při návrhu úložiště s tím je třeba počítat. Jsme tedy zpátky u kopírovacího konstruktoru. Pokud se síť změní a my nevíme jak, je třeba ji uložit celou. Speciální případ nastává, pokud se v síti mění jen pozice jednotlivých uzlů. Potom je možno ponechat během celého výpočtu jedinou objektovou strukturu objektu *FVMMesh*, tedy objekty *Vertex*, *Volume* a *Edge* a ukládat pouze spolu s řešením $\boldsymbol{w}_i$ také nové pozice x a y u uzlů. Tento speciální případ jsme v prvotním návrhu neuvažovali, spadal by do kategorie proměnné sítě.

Prvotní objektový návrh tedy byl takový, že základem bude rozhraní *IRepository*, které budou implementovat dvě třídy *FixedMeshRepository* a *VariableMeshRepository*. *FixedMeshRepository* nechť ukládá v každé iteraci pouze dvojrozměrné pole W-hodnot a druhý *VariableMeshRepository* celou kopii do seznamu objektů *FVMMesh*. Průměrné hodnoty z tabulky 4 však názorně ukazují obrovské rozdíly v nárocích na čas a paměť obou druhů úložišť. *FixedMeshRepository* byl přibližně 20x rychlejší a 10x méně paměťově náročný než *VariableMeshRepository*. Myšlenky na to, přímo kopírovat celé objekty *FVMMesh* kopírovacím konstruktorem, jsme se tak vzdali. Bylo potřeba mít síť rekonstruovatelnou z minima uložených dat.

[5]Data vznikla testováním jako průměr ze tří výpočtů o deseti iteracích. Síť tvořilo 20 000 uzlů. Pro podrobný popis testu viz 3.5.5.

3.3 Analýza

3.3.1 Typ sítě

Rozdělíme nejdříve síť podle toho, jak se bude v průběhu výpočtu měnit její topologie :

Fixed Topologie se nemění.

Moving Měnit se mohou jen pozice uzlů.

NonFixed Topologie se může měnit libovolně.

Do objektu *FVMMesh* přidáme jako parametr typ sítě. Při ukládání sítě do úložiště bude typ rozhodovat o tom, jaké parametry bude obsahovat obraz sítě.

3.3.2 Obraz sítě

Z obrazu sítě bude možno kdykoliv zrekonstruovat identickou kopii originálu. Navrhneme jej tak, abychom pro daný typ sítě minimalizovali paměťové nároky. Obraz se vytváří během výpočtu na příkaz uživatele k uložení stavu objektu *FVMMesh*. Jeho součástí tak bude i informace o číslu iterace a času, kdy se tak uskutečnilo. Pokud bude síť typu *Fixed*, uloží se do obrazu jen vypočtená data, pokud typu *Moving*, uloží se kromě toho nové souřadnice vrcholů, které změnily pozici. V případě typu sítě *NonFixed* se ukládá celá topologie.

Speciálním případem bude obraz sítě v hlavičce úložiště. Ten nám bude sloužit jako referenční obraz k rekonstrukci sítě typu *Fixed* nebo *Moving*. Jeho struktura bude stejná, jako v případě obrazu *NonFixed* v úložišiti, jen bez čísla iterace a času. Referenční obraz v hlavičce totiž budeme potřebovat pro rekonstrukci jakékoli iterace, protože bude obsahovat celou topologii sítě.

3.3.3 Datová struktura úložiště

Úložiště navrhneme jako samostatný objekt *Repository* (viz UML diagram tříd v příloze B.2, podrobněji níže). Jeho jádrem bude datová struktura, do které budeme ukládat obrazy sítě v dané iteraci a času výpočtu. Protože budeme vyhledávat především na základě čísla iterace, bude touto datovou strukturou mapa `Dictionary<int, MeshSnapshot>`.

Kromě čísla iterace bude dalším způsobem vyhledávání v této mapě také časová značka typu double. Uživatel bude požadovat síť v libovolném čase výpočtu. Lze předpokládat, že to nebude přesně ten čas, pod kterým byla síť uložena. Iterujeme proto přes celou mapu, sledujeme časovou značku, která je

parametrem objektu *MeshSnapshot* (viz 3.4.2), a navracíme nejbližší později uložený obraz.

3.3.4 Hlavička úložiště

Existují informace, které se během celého výpočtu nemění, a nemusí se tedy ukládat do každé iterace. U všech typů sítě jsou to okrajové podmínky.[6] U typů *Fixed* nebo *Moving* potom i topologie sítě. Síť se v těchto případech rekonstruuje z referenčního obrazu v hlavičce a teprve potom se k jejím jednotlivým konečným objemům přidají data z obrazu v datové struktuře úložiště. Uveďme si názorný příklad, jak funguje takové oddělení v případě sítě typu *Moving*. Ta je algoritmicky nejsložitější a po jejím pochopení bude jasný i analogický postup pro sítě typu *Fixed* a *NonFixed*.

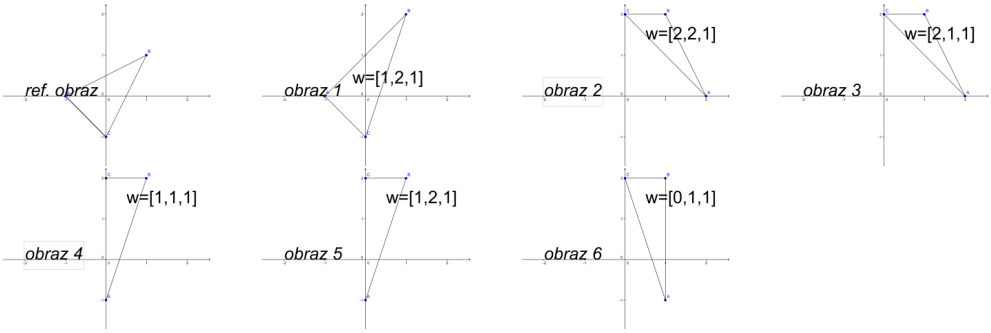
3.3.5 Ukládání a načítání sítě typu *Moving*

V případě sítě typu *Moving* je možné ušetřit v obrazu sítě místo tak, že v každé iteraci uložíme místo celé topologie pouze nové souřadnice posunutých uzlů. Počáteční topologii získáme z hlavičky sítě.

	hlavička
	ref. obraz
uzel A:	x=-1, y=0
uzel B:	x=1, y=1
uzel C:	x=0, y=-1

	datová struktura úložiště					
	obraz 1	obraz 2	obraz 3	obraz 4	obraz 5	obraz 6
uzel A:		x=2		x=0, y=-1		x=1
uzel B:	y=2					
uzel C:		y=2				
řešení:	w=[1,2,1]	w=[2,2,1]	w=[2,1,1]	w=[1,1,1]	w=[1,2,1]	w=[0,1,1]

Tabulka 5: Úložiště s obrazy sítě typu Moving



Obrázek 14: Zobrazení uložených stavů sítě

Na začátku máme v síti pro jednoduchost jen 3 uzly A[-1,0], B[1,1] a C[0,-1], které dohromady budou tvořit jediný konečný objem. Na tom vypočteme řešení $\boldsymbol{w}_1^k$, které budeme ukládat do obrazů.

[6] Okrajová podmínka je funkce, změna u ní může nastat, pokud změníme parametr.

Ukládání Při ukládání sítě typu *Moving* poskytne seznam uzlů, které oproti naposledy uložené iteraci změnily svou pozici, přímo uživatel. On sám totiž topologii sítě změnil ve svých obsluhách událostí. Stav úložiště po uložení šesti sítí představujeme čtenáři v tabulce 5 a jejich grafickou podobu na obr. 14.

Načítání Při načítání z úložiště iteruje příslušná metoda přes jednotlivé obrazy v datové struktuře a přepočítává pozici uzlů od první až k požadované iteraci. Na obr. 14 uvádíme názorný příklad. Požadujeme načíst síť z obrazu 6. Postupujeme takto:

1. Načtěme referenční topologii z hlavičky a síť zrekonstruujeme.
2. Iterujeme přes datovou strukturu úložiště až do obrazu 6 a na síti měníme souřadnice uzlů.
   - V obrazu 1 změní uzel B svou pozici na [1, 2].
   - Uzel A změní svou pozici několikrát a v obrazu 6 bude na [1, -1].
   - Uzel C taktéž a bude na [0, 2].
3. Konečné objemy sítě inicializujeme vypočtenými daty z obrazu 6 (viz 3.3.4).

Pokud bude uživatel požadovat z úložiště síť na základě časové značky, je postup analogický (viz 3.3.3).

3.3.6 Odkládání úložiště na disk

I při velkém šetření je třeba počítat s tím, že časem nebude možno celé úložiště uchovávat v paměti. Výpočty jsou náročné, mají řádově $10^3 - 10^5$ iterací a mohou trvat i několik dní. Úložiště budeme proto automaticky odkládat na disk podobným způsobem, jakým funguje stránkování paměti. Použijeme k tomu standardní serializační metody, které programovací jazyk poskytuje.

Proces odkládání Budeme muset odložit jednak hlavičku úložiště, jednak datovou strukturu s obrazy sítě. Tu však není výhodné uložit celou do jednoho binárního souboru. Zpravidla totiž nebudeme potřebovat z disku načíst naráz všechny uložené sítě, nýbrž jen síť v jedné konkrétní iteraci, od které spustíme nový výpočet. Načítání všech obrazů sítě by způsobilo příliš velkou prodlevu. Datovou strukturu proto rozdělíme na části, které budeme posléze ukládat do samostatných souborů. Každému úložišti pak bude příslušet jeden adresář

s těmito soubory. Odkládáme do něj vždy všechny obrazy v úložišti a jimi zabranou paměť posléze celou uvolníme.

Hlavičku odkládáme v témž adresáři do souboru *header.bin* (viz CD). Tento soubor je serializovaný objekt *RepositoryHeader*, jehož součástí je topologii sítě, dále okrajové podmínky pro výpočet a odkazy na odložené soubory (viz 3.4.4).

Uživatel nastavuje cestu k adresáři úložiště, počet iterací, po kterých se má úložiště odložit, a maximální počet obrazů sítě v jednom souboru.

Proces načítání při aktivovaném odkládání

1. Prohledá se datová struktura úložiště v paměti.
2. Pokud není požadovaná iterace nalezena, prohledá se hlavička úložiště a zjistí se název souboru, do kterého byla odložena.[7]
3. Soubor se načte, seznam s objekty obrazů sítě se připojí do úložiště v paměti.
4. Zrekonstruuje se topologie sítě buď přímo na základě informací z obrazu (pokud byl typu *NonFixed*), nebo z referenčního obrazu v hlavičce (pokud byl obraz v dané iteraci typu *Fixed*).
5. Ke konečným objemům se přiřadí vypočtená data z obrazu sítě.
6. Navrátí se požadovaná síť.

Problém při odkládání sítě typu Moving Uvažujme následující scénář: Máme síť typu *Moving*, výpočet proběhl ve stu iteracích. V každé iteraci se přitom posunuly určité uzly na souřadnice, které uchováváme v obrazu sítě. Úložiště jsme odložili do pěti souborů po dvaceti obrazech a paměť je nyní prázdná. Požadujeme načíst 99. iteraci.

Pokud by bylo úložiště v paměti, iterovali bychom od začátku datové struktury úložiště přes všechny iterace a konstruovali *NonFixed* síť s konečnou pozicí uzlů v 99. iteraci (viz 3.3.5).

Kdybychom však chtěli tímto způsobem rekonstruovat 99. iteraci z disku, museli bychom z něj načíst všechny předchozí iterace, tedy všech pět souborů. To i když sama 99. iterace je jen v tom posledním, pátém. Načteme-li jen jej, nemáme informaci o tom, které uzly se do 80. iterace posunuly. Známe jen

[7] Pokud požadujeme síť na základě časové značky, prohledá se jen paměť, načítání z disku jsme z časových důvodů neimplementovali.

počáteční topologii z hlavičky. Při větším počtu uložených sítí by náročnost takové operace ještě přímo úměrně rostla.

Rozhodli jsme se proto obrazy sítě typu *Moving* ještě před odložením na disk převést na *NonFixed*. Zkompletujeme tak topologii všech obrazů. Naroste sice velikost souborů, ale urychlí a zjednoduší se načítání (viz tabulka 6 níže[8]).

	čas načtení 1. iterace [s]	čas načtení 10. iterace [s]
odložení jako *NonFixed*	2,920	2,974
odložení jako *Moving*	3,349	14,370

Tabulka 6: Porovnání časů načítání sítě typu *Moving* z odloženého úložiště při předchozí konverzi na *NonFixed* a bez ní

3.3.7 Vazba úložiště a výpočtu

Základním použitím úložiště však je umožnit od určitého stavu sítě znovu spustit výpočet (viz případy užití v kapitole 3.1). Úložiště by tak mělo být pevnou součástí výpočtu, resp. jeho parametrem. K výpočtu uživatel přistupuje v obsluhách událostí přes objekt *EFVMSolverContext* (viz 2.5.4). Stejně tak by měl tedy přistupovat i k úložišti. Objekt jsme proto doplnili o odkaz na úložiště a o příkaz *Snapshot()* pro rychlé uložení aktuálního stavu sítě, která je součástí objektu kontextu také. Úložiště pak bude uživatel konfigurovat v konfiguračním souboru výpočtu.

Lze si představit i situaci, kdy budeme potřebovat založit objekt úložiště samotný, nezávisle na výpočtu. Např. budeme chtít načíst z úložiště na disku síť z jedné konkrétní iterace a řešení na této síti vizualizovat. Úložiště tak musí být konfigurovatelné i přímo v konstruktorech.

[8]Test byl proveden simulací v programu, který již odkládal síť typu *Moving* jako *NonFixed* (viz 3.5.8). Výsledné hodnoty jsou průměrem ze tří měření na souborech, které obsahovaly dvě sítě o 20 000 uzlech.

3.3.8 Další úpravy stávajícího programu

Při těsné vazbě úložiště na výpočet by bylo vhodné, aby uživatel mohl nastavit parametry úložiště přímo při konfiguraci výpočtu (viz 2.5.3). XML definici tedy bude tedy nutné obohatit. Zavedeme element <*repository*>, který bude potomkem kořenového elementu, a tedy na stejné úrovni jako elementy <*solver*> a <*solution*>. V něm bude nutno definovat:

1. zda se má úložiště načítat z externího umístění a pokud ano, cestu k příslušnému adresáři
2. zda je povoleno odkládání a pokud ano:
   (a) adresář pro odkládání
   (b) počet iterací v jednom souboru
   (c) počet iterací, po kterých má být úložiště odloženo

3.4 Objektový návrh

3.4.1 MeshType

Typ sítě (viz 3.3.1) implementujeme jako výčtový datový typ *MeshType*.

Třída 1: <<enumeration>>MeshType

```
Fixed
Moving
NonFixed
```

3.4.2 MeshSnapshot

Objekt *MeshSnapshot* reprezentuje obraz sítě (viz výše v kapitole 3.3.2) v určité iteraci výpočtu. Budeme se snažit maximálně šetřit pamětí a používat pouze primitivní datové typy a pole. Prvky, které spolu souvisejí, proto také neukládáme do samostatných obalovacích objektů, jak je v objektově orientovaném programování běžné, nýbrž vedle sebe. Jejich propojení bude pouze na základě indexu v polích.

Třída 2: MeshSnapshot

```
+W : double[][]
--
+movedVerticesIndexes : int[]
+movedVerticesNewXs : double[]
+movedVerticesNewYs : double[]
--
+xs : double[]
+ys : double[]
+indexBindexes : int[]
+indexBs : int[]

+volumeVertices : int[][]
+volumeData : double[][]
+regionIndexes : int[]
+regions : int[]

+boundaryEdgesVert1 : int[]
+boundaryEdgesVert2 : int[]
+boundaryEdgesBindexes : int[]
```

W představuje pole řešení na konečných objemech, *movedVerticesIndexes*, *movedVerticesNewXs* a *movedVerticesNewYs* (zkráceně *movedVertices-*) jsou pole nových pozic uzlů pro síť typu *Moving*, *xs* a *ys* jsou pole pozic uzlů, *indexBindexes* a *IndexBs* označují index okraje, na kterém aktuální uzel leží, *volumeVertices* propojení uzlů do konečných objemů, *volumeData* označuje doplňková data na konečných objemech, *regionIndexes* a *region* pole oblastí, pole celých čísel *boundaryEdgesVert1*, *boundaryEdgesVert2* a v neposlední řadě i*boundaryEdgesBindexes* (zkráceně *boundaryEdges-*) označují okrajové hrany.

Hodnoty řešení ukládáme do dvojrozměrného pole *W*. Jeho první složka zde ale má jiný význam, než v původním programu, kde 0 označovala staré řešení a 1 aktuální (viz 2.5.2). Zde je první složkou index konečného objemu i. Druhá složka je v obou případech komponenta řešení vektoru $\boldsymbol{w}_i$. Pro všechny typy sítě budeme do úložiště ukládat tentýž objekt obrazu. Rozdíl bude pouze v inicializovaných polích.

- *Fixed* síť ukládá jen pole *W*.
- *Moving* síť ukládá navíc indexy uzlů, které oproti předchozí uložené iteraci změnily svoji pozici, a nové souřadnice do polí *movedVertices*-. Stejný index v těchto třech polích značí tentýž uzel.
- *NonFixed* síť neukládá pole *movedVertices*-, ale všechna ostatní zbylá, ze kterých bude možno kompletně rekonstruovat topologii sítě, tj.:
  1. Pozice uzlů v polích *xs* a *ys*, jejichž index odpovídá indexu uzlu. Dále informaci, které z nich jsou okrajové (pole *indexBindexes*) a index okraje, ke kterému tyto příslušejí (pole *indexBs*).
  2. Propojení uzlů do konečných objemů (pole *volumeVertices*). První složkou je zde index konečného objemu a druhou to, kolikátý v pořadí je to jeho uzel. Protože máme ze zadání troj- nebo čtyřúhelníkovou síť, bude pole délky 3 nebo 4. Pro kompletní rekonstrukci konečných objemů potřebujeme ještě doplňková data (pole *volumeData*) a označení, které uzly patří do některé předem specifikované oblasti (pole *regionIndexes*) a do které (pole *regions*)
  3. Hrany, které jsou na okraji a na kterém (tři pole *boundaryEdges*-). Na index okraje pak odkazuje pole *indexBs* u uzlů.

3.4.3 BoundarySnapshot

Podobně jako v případě objektu *MeshSnapshot* budeme potřebovat ukládat také obraz okrajových podmínek. Ty jsou společné pro celý výpočet. Objekt *FVMMesh* uchovává odkazy na uzly, hrany a objemy sítě. Jejich část, která se týká aktuální okrajové podmínky, uchovává také objekt *Boundary* (viz diagram tříd v příloze B.1). Při rekonstrukci objektu *FVMMesh* z obrazu sítě však vznikají tyto zanořené objekty nově a ty, které zůstanou v objektu *Boundary*, jsou tak zastaralé. Pokud bychom objekt *FVMMesh* serializovali, zároveň s ním by se automaticky serializovaly také tyto nepoužívané objekty uvnitř něj.

Vytvořili jsme tedy objekt *BoundarySnapshot*, který (až na výjímku) ukládá primitivní datové typy. Odkazy na hraniční objekty ukládat nebude. Předpokládá se, že napojení bude doplněno při rekonstrukci sítě z jejího obrazu na základě parametru *IndexB* uzlu, hrany či objemu.

Třída 3: BoundarySnapshot

```
+BoundaryDelegate : BoundaryFunction
+ID : int
+Type : BoundaryType
+TMin : double
+TMax : double
+CbID : int
+FixedPrescribedData : double[];
+VariablePrescribedData : Func<Vector2D, double, double↩
    >[];
```

BoundaryDelegate je funkce, která určuje tuto okrajovou podmínku, *TMin* a *TMax* její minimální a maximální parametr, *ID* identifikátor, *Type* typ, *CbID* index propojené hranice, *FixedPrescribedData* data okrajové podmínky a *VariablePrescribedData* proměnná data na hranici $f = f(x, y, t)$.

3.4.4 RepositoryHeader

Třída 4: RepositoryHeader

```
+RepoType : MeshType
+Boundaries : List<BoundarySnapshot>
+RefNonfixedSnapshot : MeshSnapshot
-_serializedIterations : Dictionary<int, string>
-------------------------------
+SerializedIterationFileName(int iteration) : string
+AddSerializedIteration(int iteration, string fileName)
+DeleteGreaterIterations(int firstIteration)
```

Data sítě, která se napříč iteracemi nebudou měnit, oddělíme do objektu *RepositoryHeader*, reprezentujícího hlavičku úložiště (viz 3.3.4). Při výpočtu na síti, která je typu *Fixed* nebo *Moving* se použije hlavička pro uložení referenčního obrazu sítě(*RefNonfixedSnapshot*). Tu potom načítáme ve chvíli, kdy síť znovu rekonstruujeme. Vypočtené hodnoty se oproti tomu načtou z obrazu sítě v úložišti. V případě typu sítě *NonFixed* (obecné změně topologie) se z obrazu v úložišti načítá jak topologie, tak řešení. Kromě referenčního obrazu sítě ukládá hlavička také obrazy okrajových podmínek. Posledním důležitým parametrem hlavičky je mapa iterací odložených na disk. Abychom

mohli reprodukovat stav úložiště, bude potřeba při jeho odkládání na disk odložit i samotnou hlavičku.

SerializedIterationFileName navrací jméno souboru, ve kterém je uložena hledaná iterace.

AddSerializedIteration přiřazuje název souboru ke konkrétní iteraci do mapy *_serializedIterations* v hlavičce. Předpokládá, že iterace již byla do uvedeného souboru odložena.

DeleteGreaterIterations smaže z disku všechny soubory s itreacemi, které jsou větší, než zadaná. Zároveň na ně smaže všechny odkazy z mapy *_serializedIterations*.

3.4.5 RepositoryMeshWrapper

Třída 5: RepositoryMeshWrapper

```
+Mesh : FVMMesh
+Config : ExplicitSolverConfiguration
+LoadedIteration : int
+LoadedTime : int
```

Objekt *RepositoryMeshWrapper* obaluje do kompaktního celku síť, která byla načtena z úložiště. K ní je přiřazeno její číslo iterace, čas ve výpočtu a také konfigurační soubor výpočtu, při kterém byla data na síti spočtena.

3.4.6 Repository

Úložiště reprezentuje třída *Repository*. Jejím klíčovým atributem je mapa *_repo*, neboť právě do něj se ukládají obrazy sítě v závislosti na iteraci výpočtu. V této třídě jsou také metody pro odkládání, ty automatické i ty jednorázové. Objekt úložiště je volně provázán s objektem výpočtu (viz diagram tříd v příloze B.2).

Třída 6: Repository

```
-_repo : Dictionary<int, MeshSnapshot>
-_config : SolverConfiguration
-_repoHeader : RepositoryHeader
-_loadingRepoHeader : RepositoryHeader
_____________________________________
+Empty() : bool
+Clear()
+UpdateConfig(ExplicitSolverConfiguration config, int ↩
    iteration)
+LoadLast() : RepositoryMeshWrapper
+LoadTime (double ptime) : RepositoryMeshWrapper
+LoadIteration(int iteration) : RepositoryMeshWrapper
+Save(FVMMesh mesh, ExplicitSolverConfiguration config, ↩
    int iteration, double time, Vertex[] movedVertices = ↩
    null)
+SerializeRepo()
+SerializeRepo(int iterationsPerFile)
+DeserializeRepo(String repoDirectory, int fromIteration,↩
     int toIteration)
```

Empty() navrací, zda je úložiště prázdné (v paměti i na disku).

Clear() vyprázdní úložiště v paměti.

UpdateConfig(...) aktualizuje konfigurační soubor, který se bude přidávat při uložení sítě do jejího obrazu. Smaže přitom všechny obrazy z větších iterací, než zadaná, čímž úložiště připraví pro nový výpočet.

LoadLast() navrací síť, rekonstruovanou z obrazu, který byl do úložiště uložen naposled.

LoadTime(double ptime) navrací síť, rekonstruovanou z obrazu, jehož čas je nejbližší vyšší vůči zadanému. Pro jednoduchost prochází pouze úložiště v paměti.

LoadIteration(int iteration) navrací síť, rekonstruovanou z obrazu v dané iteraci. Pokud se nevyskytuje v paměti, deserializuje nejprve soubor s příslušnou iterací a celou jeho část úložiště připojí k již existujícímu v paměti. Potom danou síť navrátí.

Save(...) vytvoří ze sítě její obraz, který následně uloží do úložiště. Pokud bylo v konstruktoru nebo konfiguračním souboru nastaveno odkládání a byla překročena mez *iterationsTillSerialization*, odloží se celé úložiště z paměti na disk. Parametry:

FVMMesh mesh síť pro uložení

ExplicitSolverConfiguration config konfigurační soubor, při kterém síť vznikla

int iteration číslo iterace výpočtu

double time čas výpočtu

Vertex[] movedVertices seznam uzlů, které od poslední uložené sítě v úložišti změnily svou polohu. Ten poskytne uživatel sám, který pohyb uzlů určuje. Parametr je relevantní pouze v případě sítě typu *Moving*.

SerializeRepo() odloží úložiště na disk. Adresář pro uložení a počet iterací na jeden soubor jsou buďto z konstruktoru objektu nebo z konfiguračního souboru.

SerializeRepo(int iterationsPerFile) funguje stejně, jen je počet iterací na soubor parametrem metody.

DeserializeRepo(String repoDirectory, int fromIteration, int toIteration) načte určitý úsek úložiště z disku do paměti. Tento úsek může být uložen i ve více souborech.

3.5 Použití

Jak se naše rozšíření programu bude používat v praxi si představíme na testech, které budou odpovídat jednotlivým případům užití (viz 3.1). Všechny probíhaly nad týmž problémem z oblasti mělké vody (viz 2.1.3).

3.5.1 Ukládání a načítání

Nejdříve v testu *TestLoading* otestujeme ukládání stavu sítě (viz bod 1), její zpětné načtení na základě čísla iterace (bod 2) a na základě času (bod 3).

Kód 10: TestLoading

```
1 EFVMSolver huygensSolver = SweHuygensFlatSolve(@"↩
    ../../../../Meshes/huygens2.msh", "repotests/↩
    huygensconfig.xml");
2 huygensSolver.Solve();
3 ...
4 RepositoryMeshWrapper loadedMeshData = solver.Repository.↩
    LoadIteration(resumedIteration);
5 solver.Solve(loadedMeshData);
6 ...
7 const double TEST_TIME = 0.014;
8 RepositoryMeshWrapper rmw = huygensSolver.Repository.↩
    LoadTime(TEST_TIME);
9 huygensSolver.Solve(rmw);
```

V tomto případě jsme požadovali čas 0.014 a úložiště nám navrátilo síť v nejbližším vyšším čase 0.02 (ověření viz test na CD). Výpočet navrací metoda *SweHuygensFlatSolve*, která načte síť ze souboru, nastaví jí počáteční podmínky (viz 2.5.2), inicializuje nad ní výpočet s danou konfigurací a nastaví mu obsluhy událostí (viz 2.5.4). V konfiguračním souboru výpočtu na adrese *repotests/huygensconfig.xml* jsme nejprve aktivovali úložiště pomocí elementu <repository /> (viz 3.3.8). Nad stavem výpočtu a ukládáním stavu sítě do úložiště má uživatel kontrolu přes obsluhy událostí a objekt *EFVMSolverContext* (viz kód 11).

Kód 11: Obsluha události AfterComputationalStep

```
1 static void AfterComputationalStep (EFVMSolverContext ↩
      context)
2 {
3   ...
4   context.Snapshot();
5 }
```

Uživatel rozhoduje o tom, kdy se aktuální stav sítě bude do úložiště ukládat. V našem případě se ukládá v každé iteraci, obecně to ale tak být nemusí.

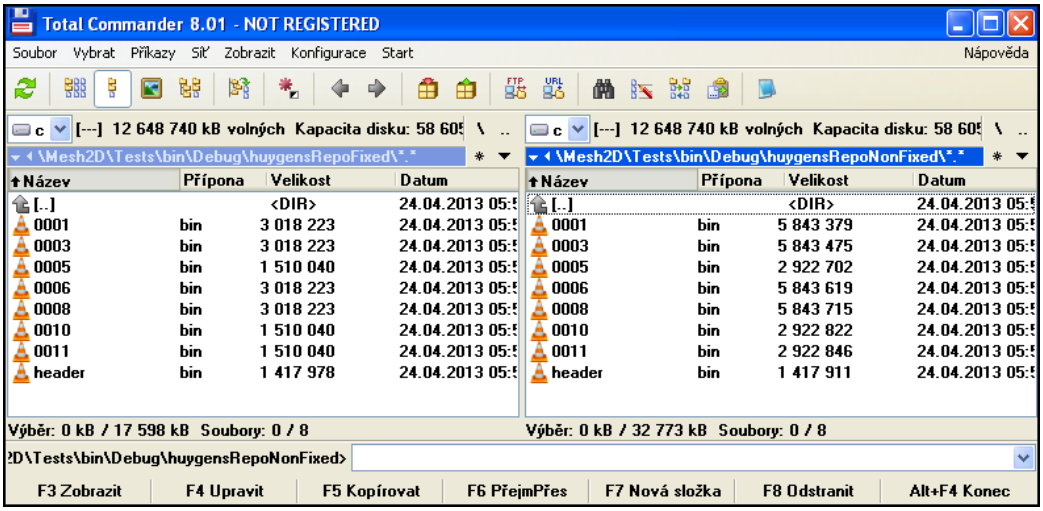
3.5.2 Odkládání na disk

Úložiště můžeme dále v elementu <repository> konfigurovat tak, aby bylo automaticky odkládáno na disk (případ užití 5). *TestLoadingWithSwapping* se nebude lišit od *TestLoading* (viz 3.5.1), rozšíří se jen konfigurační soubor (viz kód 12).

Kód 12: Rozšíření konfigurace výpočtu o konfiguraci úložiště

```
1 <repository>
2   <swapping>
3     <directory>huygensRepo1</directory>
4     <iterationsTillSerialization>5</↩
          iterationsTillSerialization>
5     <iterationsPerFile>2</iterationsPerFile>
6   </swapping>
7 </repository>
```

Při tomto nastavení se každých pět iterací se pole obrazů sítí *_repo* (viz 3.4.6) rozdělí na části po nejvýše dvou (tedy na 2, 2 a 1), každá se uloží do samostatného souboru a pole *_repo* se následně vyprázdní. Po skončení výpočtu se zbylé obrazy uloží na disk také. Stav adresáře spolu s velikostmi jednotlivých souborů poté uvádíme na obr. 15. Na jednu síť o 20 000 uzlech, připadlo přibližně 1,5 MB. To za předpokladu, že síť byla typu *Fixed* a její topologie se neměnila. Při uložení kompletní topologie do každého obrazu sítě naroste i velikost souborů přibližně na dvojnásobek. Kromě souborů se stavy sítě v jednotlivých iteracích byl uložen hlavičkový soubor *header.bin*.



Obrázek 15: Adresář úložiště

3.5.3 Načítání z externího úložiště

V testu *TestExternalLoading* testujeme načítání z externího úložiště (viz případ užití 6). Definujeme elementu <repository> atribut *externalLoadDir*.

Načítání pak probíhá postupem stanoveným v kapitole 3.3.6: Pokud není nalezena iterace v paměti, načte se ze souboru header.bin hlavička úložiště, která obsahuje seznam iterací, odložených na disk. V tomto seznamu se vyhledá jméno souboru, z kterého načteme obrazy sítě a uložíme je do datové struktury úložiště. Načtené úložiště je pak možné opět serializovat na jiné místo, v našem případě to však nevyužíváme.

3.5.4 Změna konfiguračního souboru

Otestovali jsme (*TestUpdateConfig*), že v libovolné iteraci výpočtu je možno změnit konfigurační soubor. Ten může mít jiné parametry, které budou poté dále ovlivňovat průběh výpočtu. Aktualizaci provedeme v aktuálním kroku výpočtu a můžeme jej znovu spustit buď odtud, nebo od libovolné předchozí iterace (viz 3.1, bod 4). V našem případě má nový konfigurační soubor odlišné hodnoty maximálního časového kroku a CFL (viz 2.5.3). Samotný test má tento průběh:

1. Založíme a spustíme celý výpočet s konfiguračním souborem *config1* za zapnutého odkládání.

2. Změníme konfigurační soubor na *config2* a spustíme výpočet od iterace 6.

3. Založíme nový výpočet s konfiguračním souborem *config2*, který bude načítat z odloženého úložiště z bodu 1. Spustíme jej od iterace 6.

4. Změníme konfigurační soubor na *config1* a spustíme výpočet z bodu 3 od iterace 6.

Výsledné hodnoty z bodů 1 a 4 jsou totožné, stejně jako z bodů 2 a 3. Změna konfiguračního souboru tedy funguje správně.

3.5.5 Kopírovací konstruktor *FVMMesh*

Testujeme kopírovací konstruktor podle původního návrhu, popsaného v části 3.2. Tento test jsme vytvořili čistě pro získání dat, která potvrzují nevhodnost tohoto řešení. Pro uložení stavu sítě doporučujeme uživateli používat metody samotného úložiště *Repository*. Test jsme rozdělili na dvě části. Testujeme jednak správnost kopie a jednak časové a paměťové nároky.

- Správnost kopie (*TestCopyConstructorFunctionality*)

  1. Načteme síť ze souboru a provedeme výpočet.
  2. Tutéž síť načteme ještě jednou do jiného objektu výpočtu. Nastavíme, že se má přerušit po 4. iteraci, a spustíme jej.
  3. Provedeme kopii objektu sítě *FVMMesh* a aktualizujeme jí druhý objekt výpočtu.
  4. Pokračujeme ve výpočtu.

  Pokud jsou vypočtené hodnoty z bodů 1. a 4. stejné, úspěšně jsme správnost algoritmu kopírování pro tento případ otestovali.

- Časové a paměťové nároky (*TestCopyConstructorTimeMemory*)

  V tomto případě bude zapotřebí změnit obsluhu událostí. Postup bude následující:

  1. Provedeme výpočet a průběžně ukládáme sítě typu *Fixed* do úložiště. V obsluze *AfterComputationalStep* standardně zavoláme

     ```
     context.Snapshot();
     ```

     a po skončení výpočtu vypíšeme nároky na zdroje.
  2. Nasimulujeme si úložiště jako globální seznam

     ```
     List<FVMMesh> _testCopyRepo = new List<FVMMesh>();
     ```

     Nastavíme výpočtu jinou obsluhu události AfterComputationalStep, ve které budeme ukládat do zmíněného seznamu kopii aktuální síť pomocí

     ```
     _testCopyRepo.Add(new FVMMesh(context.Mesh));
     ```

     Na konci výpočtu opět obdržíme průměrná data.

Výsledky uvádíme v tabulce 4 na str. 35. Získali jsme tím hodnoty, které potvrzují doměnku, že hluboká kopie má příliš velké nároky. V této práci s ní již dále nepočítáme.

3.5.6 Animace

Test animace (*TestAnimation*) provedeme pomocí exportu jednotlivých iterací z úložiště. Ty nemusejí pokaždé bezprostředně následovat. Na začátku testu nastavíme počáteční iteraci, krok a počet generovaných snímků. Poté iterujeme ve smyčce for a po jednom příslušné iterace z externího úložiště načítáme a exportujeme do VTK souborů. Použijeme k tomu již existující metody třídy *VTKXmlDataVisualisator* (viz 2.5.7). Výslednou sérii souborů *testanimation-...vtp* lze načíst ve výše uvedených programech pro vizualizaci (viz 2.4.6).

3.5.7 Síť typu *Moving*

V testu *TestMoving* testujeme, zda budou fungovat algoritmy pro síť, u které se v čase mění pouze pozice vrcholů (viz 3.3.5). V obsluze události BeforeConditionalStep měníme pozice uzlů, v AfterComputionalStep potom jen vizualizujeme nové řešení.

Kód 13: TestMovingBeforeComputionalStep, obsluha události

```
1  if (context.Iteration % 2 == 0) {
2    for (int i = 0; i < context.Mesh.Vertices.Count; i++)
3    {
4      Vertex vx = context.Mesh.Vertices[i];
5      Vector2D a = vx.Position;
6      if (context.Iteration % 4 == 0) {a.X = a.X * 0.8;}
7      else {a.Y = a.Y * 1.1;
8      }
9      vx.Position = a;
10     for (int j = 0; j < vx.CommonEdges.Count; j++)
11     {
12       Edge edge = vx.CommonEdges[j];
13       edge.Length = edge.ComputeLength();
14       vx.CommonEdges[j] = edge;
15     }
16     context.Mesh.Vertices[i] = vx;
17   }
18   _counter++;
19 }
```

Každou druhou iteraci tedy buď od sebe uzly vzdálíme ve směru osy y nebo přiblížíme ve směru osy x. Vizualizujeme standardním mechanismem, představeným v kapitole 2.5.7.

3.5.8 Síť typu *Moving* s odkládáním

Tento test (*TestMovingSwapping*) slouží pro porovnání časů načítání sítě, odložené v jednom případě jako *Moving* a ve druhém jako *NonFixed*. Problém podrobněji popisujeme v samostatné podčásti kapitoly 3.3.6. Jelikož bylo již implementováno kvalitnější druhé řešení, první probíhá pouze formou simulace. Postup je takovýto:

- Spustíme výpočet se zapnutím odkládáním a po jeho skončení vyčistíme úložiště v paměti.
- 1. iterace při *Moving* - Požadujeme načtení 1. iterace (soubor 1.bin), nejprve jednou, což bude představovat načtení obrazu z disku do paměti, potom podruhé kdy se bude načítat hlavička (již přímoz paměti). Budeme muset poté také připočítat čas pro rekonstrukci topologie ze sítě *Moving* (viz 3.3.5).
- 1. iterace při *NonFixed* - Vyčistíme úložiště a pouze načteme iteraci 1 z disku. Režie pro rekonstrukci sítě typu *Moving* odpadá.
- 10. iterace při *Moving* - Požadujeme načtení 10. iterace (soubor 10.bin), při následné rekonstrukci sítě typu *Moving* pak ale musíme (analogicky k obr. 14) mít v paměti i všechny předchozí iterace od 1. Odkládali jsme po 2 sítích do jednoho souboru, načteme tedy ještě soubory 1.bin, 3.bin, 5.bin, 6.bin, 8.bin. Když již máme nyní všechny iterace v paměti, nesimulujeme nyní procházení celým tímto polem a měnění pozice uzlů. Reálný čas by byl tak ještě vyšší.
- 10. iterace při *NonFixed* - Pouze načteme iteraci 10 z disku (10.bin).

Výsledky jsou shrnuty v tabulce v kapitole 3.3.6 na str. 38.

3.5.9 Transformace dat

Test *TestRepoSolutionTransformation* bude sloužit k vygenerování lineárních transformovaných dat. Načteme síť, provedeme na ní výpočet a výsledná po částech konstantní data na konečných objemech budou posléze převedena na lineární řešení pomocí funkce, kterou programuje uživatel (v našem případě *transFunction*, která navrátí součet převrácených hodnot všech složek řešení).

V tomto testu selhala z neznámého důvodu transformace na referenčním výpočtu (viz 3.7). Používáme proto již jeden z připravených výpočtů pro jiné problémy (viz následující kapitola). Výsledná transformovaná data budou zobrazena na 2D modelu sítě pomocí různé intenzity barev.

3.6 Použití v praxi

Všechny dosud představené testy pro jednotlivé případy užití byly postavené nad touž sítí. Ta simulovala problém mělké vody, popisující nestlačitelné proudění, u kterého se počítala výška hladiny a momenty hybnosti (viz 2.1.3). Nyní zvolíme opačný postup. Vybereme jeden test a budeme jej aplikovat na různé sítě. Zvolíme test nejzákladnějšího případu užití - načtení iterace, která již je v paměti a obnovení výpočtu od tohoto bodu (*TestLoading*). Testy, které již byly napsány pro původní verzi programu, obalíme naším testem pro načítání z repozitáře. Jednotlivé problémy zde blíže vysvětlovat nebudeme a pouze představíme síť, se kterou pracují. Čtenáře odkážeme na kód testů v původním programu.

1. Riemannův problém (*SweRiemannFlat1.cs*) je taktéž problémem mělké vody. Pracuje na klasické mřížce, kterou jsme již představili na obr. 3 na str. 11.

2. Vortex Evolution Problem (*VEproblem.cs*) již problémem mělké vody není. Popisuje proudění stlačitelné, u něhož se počítají hodnoty hustoty, momentů hybnosti a energie, a to pomocí Eulerových rovnic. V tomto případě nám však obnovený výpočet selhal, viz 3.7.

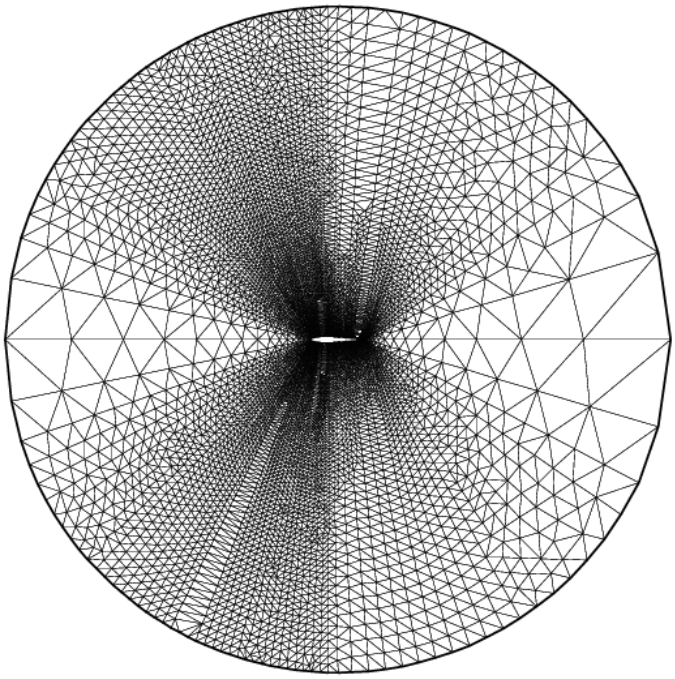
3. NACA0012 (*NACA0012.cs*) řeší proudění vzduchu kolem křídla, které je stlačitelné, a popisují jej proto Eulerovy rovnice taktéž.

   Výpočet tohoto problému skončí na maximální počet iterací, který je stanoven na 1000. První necháme proběhnout kompletně za zapnutého odkládání, ve druhém začneme od stavu v iteraci 660. Na obr. 16 je znázorněna síť, na které výpočet probíhá. Budeme sledovat různé konfigurace repozitáře, abychom zjistili, jakým způsobem závisí doba výpočtu na velikosti načítaného souboru. Zkoumáme, jaký počet iterací je vhodné odkládat do jednoho souboru, aby byla co nejkratší celková doba výpočtu, včetně času pro odkládání a načítání.

	první výpočet - celý		druhý výpočet - částečný[9]	
iterací[10]	čas serializace	celkový čas	čas deserializace	celkový čas
100	12 s	7 min 38 s	2 min 14 s	4 min 5 s
20	1,51 s	6 min 21 s	8 s	2 min 3 s
2	0,25 s	6 min 43 s	0,52 s	2 min

Tabulka 7: Porovnání časů výpočtů při rozdílně nastaveném odkládání v konfiguračním souboru

V tabulce 7 uvádíme naměřené výsledky[11]. Odkládat po 100 iteracích je špatnou volbou, neboť se pak soubor s těmito iteracemi bude příliš dlouho načítat. Optimum je v tomto případě odkládat na disk naráz přibližně po 20 iteracích. Při větším číslu se doba načítání neúměrně prodlužuje a při menším úspor nedosáhneme.



Obrázek 16: Síť problému NACA

[9] Od iterace 660 do 1000.
[10] Parametr *iterationsPerFile* v konfiguračním souboru.
[11] Test proběhl na počítači s procesorem, taktovaném na 1,83 GHz a pamětí 2GB.

3.7 Nedostatky a chyby práce

V naší práci přiznáváme tyto nedostatky, které by mohly být řešeny dále:

1. Objektová struktura nepříliš kompaktní

   Na objekt *ExplicitSolverConfiguration* je příliš mnoho vazeb, které jsou pravděpodobně nadbytečné (viz diagram tříd v přiloze B.2). Objekt *EFVMSolver*, jejž nastavujeme v XML souboru konfigurace především (viz příloha D), by mohl předávat příslušné parametry konfigurace přímo konstruktoru objektu *Repository*. Takové parametry jsou totiž nejvýše 4 - adresář pro odkládání a adresář pro načítání úložiště, počet iterací na jeden soubor a počet iterací, po kterých se odkládá. Sám objekt *Repository* by potom odkaz na konfigurační soubor nepotřeboval. V případě, že je objekt *Repository* zakládán uživatelem sám o sobě, má nyní odkaz na konfigurační soubor stejně hodnotu null. Na druhou stranu objekt *MeshSnapshot* má zatím uloženo, jaký konfigurační soubor byl použit pro výpočet aktuální iterace. Otázka je, jestli uživatel takovou informaci potřebuje, nebo zda má poskytnout vždy svůj konfigurační soubor, i např. při restartu výpočtu od určité iterace. Tento problém by proto vyžadoval důkladnější rozbor.

2. Problém s vizualizací v MVE-2. Na dvou počítačích s různými operačními systémy nám program skončil při pokusu o zobrazení dat chybou. Ta nebyla v modulu pro načtení VRML souboru, nýbrž pravděpodobně v použitém 3D-rendereru. Pomoci by mohla větší komunikace s vývojáři programu a nahlédnutí modulu pro vizualizaci, který MVE-2 používá.

3. VEproblem - při obnovení výpočtu od 660. iterace program skončí s výjímkou

   ```
   ArithmeticException Function does not accept
   floating point Not-a-Number values.
   ```

   Stane se tak v modulu QVolume, když se konstruuje čtyřúhelníkový konečný objem z uzlů o indexech 390, 391, 400 a 401. Je to způsobeno neurčitým výrazem Infinity/Infinity, který se následně vyskytne ve vzorci pro výpočet středového bodu dle Cramerova pravidla (viz 2, str. 14). Chybu jsme neopravili.

4. Formátu Exodus, který se v testech ukázal jako nejúspornější (viz tabulka 2), mohla být věnována větší pozornost. Mělo by se prozkoumat, jakou má tento binární formát strukturu, zda je vůbec někde k dispozici dokumentace, zda existují generátory a konvertory, co další programy, které by jej uměly načíst, kde se v praxi používá, atd...

5. Neodhlalili jsme, proč programy VisIt a Mayavi2 nenačtou testovací soubory formátu PLOT3D, ačkoli jeho podporu avizují. Zkoušeli jsme více souborů tohoto formátu se stejným výsledkem, proto se domníváme, že chyba musela být ve vizualizačních modulech programů.

6. Pro program Mayavi2 jsme neprozkoumali načítání dat, které by přímo generovaly pluginy Pythonu. Nebyly nahlédnuty rozdíly takového přístupu oproti prostému načítání VTK souboru.

7. Čas načítání odložené sítě typu *Moving* byl nasimulován několikerým načítáním sítě typu *NonFixed* (viz 3.5.8), jelikož byl pro odkládání automatický převod *Moving* na *NonFixed* již implementován. Pro přesné hodnoty bychom museli v jednom případě tuto konverzi vypnout a v jendom zapnout. Při načítání by pak taky nastalo takové podobné rozdělení: *NonFixed* načítat z jednoho souboru, *Moving* navíc ze všech předchozích od začátku úložiště.

8. Modul pro transformaci dat z výpočetní do grafické vrstvy, který máme ve třetím bodě zadání za úkol vytvořit, není jeden celistvý. Uživatel musí pro vizualizaci průběhu výpočtu vykonat dva kroky. Nejprv si vygeneruje síť z jejího obrazu v úložišti (objekt *Repository*), k čemuž použije námi naimplementované metody (viz 3.4.6). Poté síť s daty vizualizuje prostřednictvím tříd, které ale již byly nedílnou součástí původního programu a které zmiňujeme v kapitole 2.5.7.

9. Transformovaná data z výpočtu pro simulaci Huygensova principu se v testu *TestRepoSolutionTransformation* z neznámého důvodu nepodařilo vizualizovat. Exportujeme VTK soubor, který ale žádný námi zmíněný vizualizační software nenačtě. Transformovali jsme tedy data z jiného výpočtu, u kterého načtení souboru fungovalo. Chybu jsme neodhalili.

3.8 Další možná rozšíření

V této kapitole naznačíme další možný vývoj našeho programu. Za vhodná rozšíření považujeme:

1. GUI pro program. Mělo by umožňovat :

   (a) Výběr MSH souboru sítě.

   (b) Na základě zadaných parametrů generovat konfigurační soubor.

   (c) Zobrazit obsah úložiště, odloženého na disk, vybrat z něj konkrétní síť a tu exportovat do VTK souboru. O úroveň složitější by pak bylo, mít možnost vizualizovat síť přímo v našem programu. Sama knihovna VTK by to nicméně umožňovala.

   (d) Provádět nad vypočtenou sítí řezy. V rozšířené verzi by mohlo být také možné generovat na základě takového výřezu ještě před samotným výpočtem okrajové podmínky.

2. Data sítě ukládáme do XML souboru formátu VTK v textové ASCII podobě (viz element *<DataArray>* v příloze C.3.2). Ušetřit více místa na disku by bylo možné, kdybychom tato data ukládali v binární nebo komprimované podobě. Umožňuje to *AppendedData* sekce, na jejíž určitý offset odkazují jednotlivá datová pole (viz [11, str. 16]).

   Stálo by tedy za to prozkoumat, zda VTK knihovna sama obsahuje kodér, který do sekce přidá zkomprimovaná data spolu se zmíněnými offsety.

3. Automatické odkládání úložiště na disk podle aktuální potřeby. Úložiště bychom neodkládali na základě parametrů v konfiguračním souboru, nýbrž prostě ve chvíli, kdy dojde paměť. Vyžadovalo by to detailnější seznámení s fungováním Garbage Collectoru C#.

4. Paralelní výpočet. Pokud je zapnuté odkládání, musí se nyní výpočet před odložením celého úložiště na disk zastavit. Kdyby však tento pracoval v samostatném vlákně, mohl by pokračovat i při procesu odkládání. Pokud bychom např. ukládali každou dvacátou iteraci, bylo by dostatek času na to, aby úložiště bylo odložené, než výpočet doběhne do následujícího bodu uložení[12]. Výpočet by však také mohl odkládání předběhnout a musela by se tedy řešit synchronizace vláken.

[12] Jde o zjednodušenou formulaci!

Kromě paralelizace odkládání by bylo zajímavé také prozkoumat, jak by bylo možno paralelizovat samotný výpočet. Jak by mohla být plně využita všechna jádra CPU a jak moc by se tím výpočet urychlil.

5. Doposud jsme počítali jen s dvojrozměrnou sítí. Metoda konečných objemů však funguje na síti jakýchkoli rozměrů. Pokud by to bylo v praxi využitelné, mělo by jít poměrně snadno implementovat i síť trojrozměrnou.

6. Implementace řezu po křivce.

4 Závěr

V teoretické části jsme čtenáři představili fyzikální rovnice, které popisují proudění tekutin, a způsob jejich řešení na síti konečných objemů. Popsali jsme dále metody, jak lze vypočtená data transformovat z konečných objemů na jednotlivé uzly a jak je posléze ze sítě extrahovat podél určitého řezu.

Čtenáře jsme seznámili s původním stavem našeho programu, který tuto metodu konečných objemů implementoval. V realizační části jsme následně pro program navrhli a implementovali rozšíření o nový modul pro uložení a transformaci dat do grafické podoby.

Díky úložišti máme po výpočtu k dispozici data ze všech jeho iterací, a to za minimálních paměťových a časových nároků. Úložiště operuje v paměti a na disku, podobným způsobem, jakým funguje v operačních systémech stránkování. Odkládání úložiště může probíhat již během výpočtu, v jehož konfiguračním souboru nastaví uživatel umístění úložiště na disku a zvolí velikost jeho souborů. Odložené úložiště o správné velikosti pak může např. zálohovat na CD, přenést na jiné místo a znovu načíst. Zde může výpočet provést s jinými parametry nebo jen z úložiště bez jakékoliv vazby na výpočet pouze načíst konkrétní síť a extrahovat příslušná data. Uživateli jsme tím umožnili reprodukovat proces výpočtu a zobrazit jakýkoliv jeho krok nebo sérii kroků.

Představili jsme programy pro vizualizaci takovýchto dat a prozkoumali formáty souborů, které podporují. Navzájem jsme porovnali jejich velikosti. Pro export vypočtených dat z úložiště do grafické podoby jsme použili formát VTK. Uvedli jsme, ve kterém vizualizačním software lze sérii VTK souborů, exportovaných z úložiště, přehrávat jako animaci procesu výpočtu a generovat video.

Kromě hotového řešení jsme popsali i proces jeho vzniku a dále to, jaké byly jiné možnosti a proč se neosvědčily. Sadou testů jsme prověřili jak výsledná řešení, tak i řešení potenciální, a jejich porovnání v textu jsme opřeli o konkrétní hodnoty. Modul použili při simulaci reálných problémů z původní verze programu. Všechny případy užití modulu jsme úspěšně otestovali a práce tak splnila svůj účel.

Poděkování

Za laskavé poskytnutí obrázků děkujeme Prof. Linhartovi, za bližší představení programu MVE-2 Ing. Petříkovi a Vášovi a za vysvětlení metody nejmenších čtverců RNDr. Teskové. Za vstřícnost při zadávání externí diplomové práce děkujeme Katedře informatiky Přírodovědecké fakulty UJEP v Ústí nad Labem.

Přehled zkratek

LS Least Square

LWA Linear Weighted Average

FVM Finite Volume Mesh

EFVM Explicit Finite Volume Mesh

FF FreeFem++

MVE Modular Visualization Environment

VTK Visualization Toolkit

CFL CourantFriedrichsLewy Condition

VS Vijaysundaram Flux

SWE Shallow Water Equation

VE Vortex Evolution

NACA National Advisory Committee for Aeronautics

Reference

[1] FELCMAN, J. *Počítačové modelování*. Praha: KNM Press, 2006. 49 s.

[2] LINHART, J. *Mechanika tekutin I*. Plzeň: Západočeská univerzita v Plzni, 2009. 123 s. ISBN 978-80-7043-766-7

[3] TESKOVÁ, L. *Lineární algebra*. Plzeň: Západočeská univerzita v Plzni, 2010. 242 s. ISBN 978-80-7043-966-1

[4] HECHT, F. *FreeFem++*[online]. Third Edition, Version 3.20-3. Dostupné z: http://www.freefem.org/ff++/ftp/freefem++doc.pdf

[5] GEUZAINE, C. and REMACLE, J.-F. *Gmsh Reference Manual for Gmsh 2.7*[online]. ©8.3.2013 [cit: 12.3.2013]. Dostupné z: http://geuz.org/gmsh/doc/texinfo/gmsh.html

[6] GEUZAINE, C. and REMACLE, J.-F. *Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities*. International Journal for Numerical Methods in Engineering, Volume 79, Issue 11, pages 1309-1331, 2009

[7] PRESS, William H. *Numerical recipes in C++: the art of scientific computing*. 2nd ed. Cambridge: Cambridge University Press, ©2002. xxviii, 1002 s. ISBN 0-521-75033-4.

[8] *CourantFriedrichsLewy condition - Wikipedia, the free encyclopedia*[online]. [cit: 1.5.2013]. Dostupné z: http://en.wikipedia.org/wiki/Courant

[9] *VisIt User's Manual*[online]. ©říjen 2005 [cit: 8.4.2013]. Dostupné z: https://wci.llnl.gov/codes/visit/1.5/VisItUsersManual1.5.pdf

[10] *Getting Data Into VisIt*[online]. ©červenec 2010 [cit: 10.4.2013]. Dostupné z: https://wci.llnl.gov/codes/visit/2.0.0/GettingDataIntoVisIt2.0.0.pdf

[11] *File Formats for VTK Version 4.2*[online]. Kitware Inc. Dostupné z: http://www.vtk.org/pdf/file-formats.pdf

[12] *ParaView Manual 3.98*[online]. Kitware Inc. [cit: 8.4.2013]. Dostupné z: http://www.paraview.org/paraview/resources/software.php

[13] *ParaView Stereoscopic adjustments*[online]. [cit: 13.5.2013]. Dostupné z:
http://www.kiv.zcu.cz/en/research/downloads/product-detail-en.html?produkt_id=28

[14] *Mayavi 4.2.1 documentation*[online]. [cit: 8.4.2013]. Dostupné z:
http://docs.enthought.com/mayavi/mayavi/overview.html

[15] RAMACHANDRAN, Prabhu. *Mayavi Users Guide - Data formats*[online]. Revision 1.14. [cit: 30.4.2013] Dostupné z:
http://mayavi.sourceforge.net/docs/guide/ch03s02.html

[16] PINKAS, P. *GNUPLOT: datové soubory*[online]. ©22.8.2001 [cit: 9.4.2013]. Dostupné z:
http://www.root.cz/clanky/gnuplot-datove-soubory/

[17] *Mono*[online]. [cit: 1.5.2013]. Dostupné z:
http://www.mono-project.com/

[18] *MonoDevelop*[online]. [cit: 12.5.2013]. Dostupné z:
http://www.monodevelop.com/

[19] FRANK, Milan, VÁŠA, Libor a SKALA, Václav. *MVE-2 applied in education process.* In: .NET technologies 2006: University of West Bohemia, Campus Bory, May 29-June 1, 2006: full papers proceedings. Plzen: University of West Bohemia, 2006. s. 39-45. ISBN 80-86943-10-0

[20] KAISER, Jan. *Obecný modulární grafický subsystém pro prostředí MVE-2.* Plzeň, 2006. diplomová práce. Západočeská univerzita. Fakulta aplikovaných věd. Vedoucí práce Milan Frank.

A Uživatelská příručka

Námi navržené a implementované třídy použije programátor, který je také uživatelem. Program byl napsán v jazyce C# a požadovanou prerekvizitou pro jeho kompilaci a spuštění je tedy buď prostředí .NET na platformě Windows nebo multiplatformové Mono ([17]). Pro vývoj programu jsme použili multiplatformní opensource IDE MonoDevelop([18]), které můžeme doporučit i uživateli.

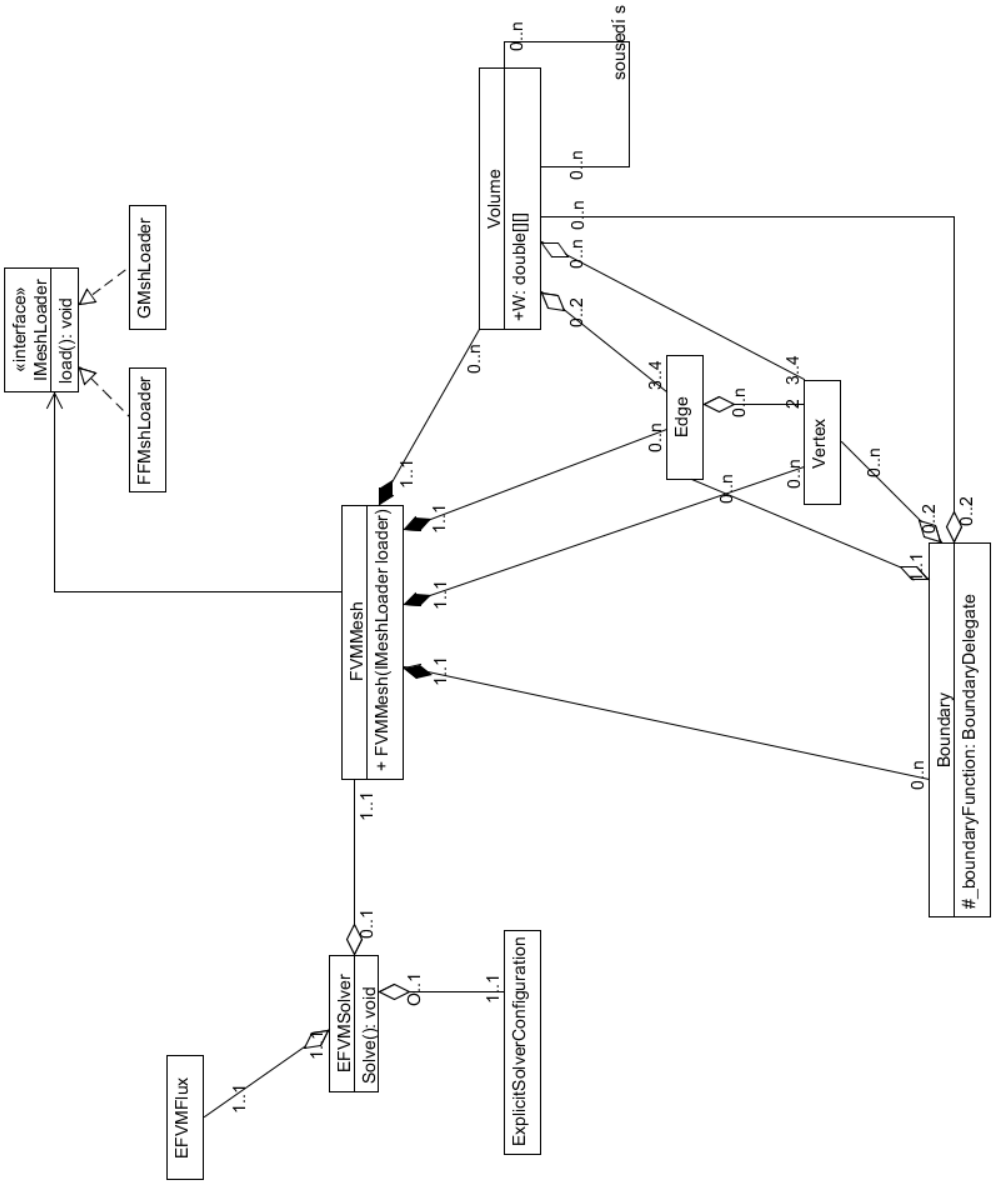
V něm otevřeme soubor *FVMSolver.sln*[13]. Aplikace obsahuje sadu projektů, z nichž je jediný spustitelný projekt *Tests* s hlavní třídou *MainClass* v modulu *Main.cs*. Zde je zakomentováno volání všech testů programu, až na sadu testů našeho úložiště, která je umístěna v modulu *RepositoryTests.cs* ve stejnojmenné třídě. V její metodě *PerformTests()* jsou všechny dostupné testy vyjmenovány, ale zakomentovány. Odkomentováním příslušného řádku spustí uživatel daný test nebo sadu testů.

Testy k případům užití jsme pro prostředí Windows předkompilovali. Uživatel je může ihned spustit z adresáře *bin/Debug* projektu *Tests*.

[13] Cestu k tomuto souboru a přesnou strukturu obsahu CD nalezne uživatel v souboru *readme.txt*.

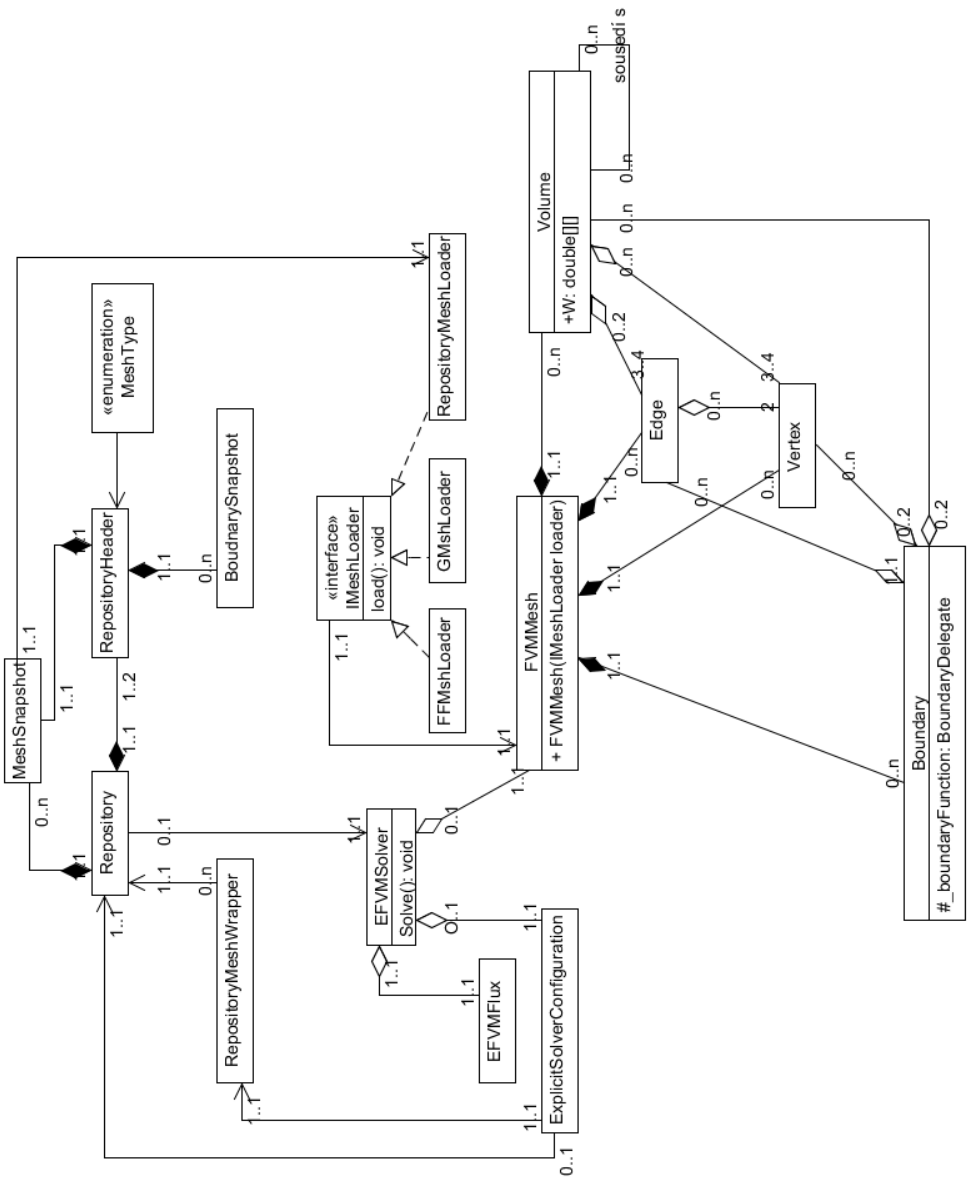
B Diagramy tříd

B.1 Původní stav programu



Obrázek 17: Diagram tříd původního programu

B.2 Konečný stav programu



Obrázek 18: Diagram tříd programu s úložištěm

C Formáty souborů

C.1 FreeFem++ MSH

```
[počet uzlů] [počet objemů] [počet okrajových hran]
(uzly)
[x] [y] [index okrajové hrany] .... (uzel 1)
[x] [y] [index okrajové hrany] .... (uzel 2)
...
(objemy)
[uzel A] [uzel B] [uzel C] [index oblasti] .... (objem 1)
[uzel A] [uzel B] [uzel C] [index oblasti] .... (objem 2)
...
(okrajové hrany)
[uzel A] [uzel B] [index okrajové hrany]
[uzel A] [uzel B] [index okrajové hrany]
...
```

Příklad (huygens.msh):

```
9864 19176 550
-10 1.5688183453e-13 1
-9.98088171018 0.618061718037 1
-9.99521928478 0.309178668533 1
...
29.9875026039 -20.4997916927 3
9864 9863 9862 0
9863 9861 9860 0
9860 9862 9863 0
...
5526 5615 5614 0
9864 9863 3
9863 9861 3
9861 9858 3
...
5267 5361 1
```

C.2 GMsh MSH

```
$MeshFormat
2.2 0 8
$EndMeshFormat
$Nodes
[počet uzlů]
1 [x] [y] [z]
2 [x] [y] [z]
...
$EndNodes
$Elements
[počet elementů]
1 [typ] 2 0 [indexB] [uzel A] ([uzel B]) ([uzel C]) ([uzel D])
2 [typ] 2 0 [indexB] [uzel A] ([uzel B]) ([uzel C]) ([uzel D])
...
$EndElements
```

Příklad (bump.msh):

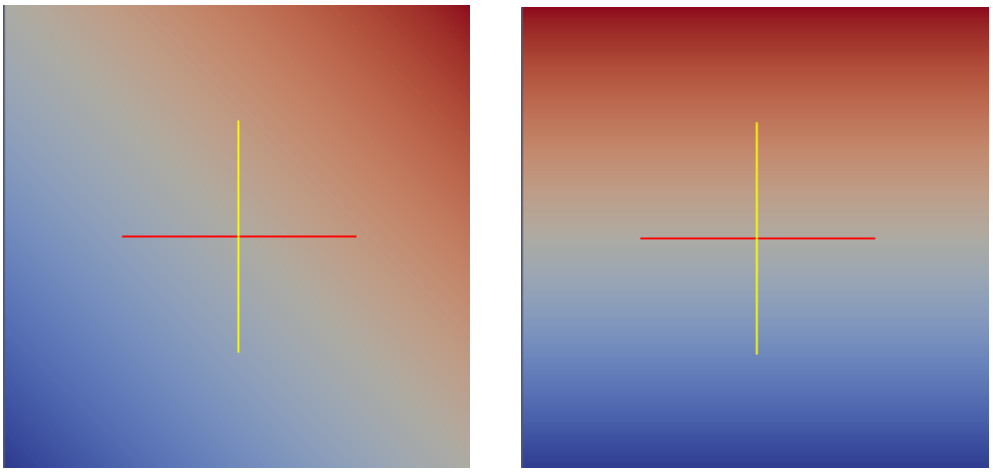
```
$MeshFormat
2.2 0 8
$EndMeshFormat
$Nodes
2747
1 0 0 0
2 1 0 0
3 2 0 0
...
$EndNodes
$Elements
3273
1 15 2 0 1 1
2 15 2 0 2 2
3 15 2 0 3 3
...
8 1 2 0 1 1 8
...
262 2 2 0 6 379 848 849
...
$EndElements
```

C.3 VTK formát

Představíme, jak bude vypadat reprezentace čtverce s dvěmi sadami dat. Souřadnice jeho vrcholů budou:

$$\begin{aligned} A[-0.5, -0.5] \\ B[0.5, -0.5] \\ C[-0.5, 0.5] \\ D[0.5, 0.5] \end{aligned}$$

$$\begin{array}{ll} MyData1: & MyData2: \\ A\{0\} & A\{0\} \\ B\{0.5\} & B\{0\} \\ C\{0.5\} & C\{1\} \\ D\{0.5\} & D\{1\} \end{array}$$



Obrázek 19: Vizualizace dat MyData1 a MyData2 z VTK souboru

C.3.1 Textová verze

```
1  # vtk DataFile Version 3.0
2  vtk output
3  ASCII
4  DATASET POLYDATA
5  POINTS 4 float
6  -0.5 -0.5 0 0.5 -0.5 0 -0.5 0.5 0
7  0.5 0.5 0
8  POLYGONS 1 5
9  4 0 1 3 2
10
11 POINT_DATA 4
12 SCALARS MyData1 float
13 LOOKUP_TABLE default
14 0 0.5 0.5 1
15 SCALARS MyData2 float
16 LOOKUP_TABLE default
17 0 0 1 1
```

Na řádku POINTS definujeme nejprve počet uzlů a poté datový typ souřadnic. O řádek níž pak postupně bet zalomení pro každý uzel vypisujeme jeho souřadnice. Ty jsou trojrozměrné, proto je třetí složka v našem případě vždy 0. Index uzlu bude záviset na tom, v jakém pořadí je zde uvedeme.
Na řádku POLYGONS definujeme počet polygonů a jejich maximální velikost. Každý další řádek představuje jeden polygon. První číslo na řádku udává počet uzlů polygonu a zbylá čísla na řádku indexy těchto uzlů.
Každá sekce SCALARS bude obsahovat data, definujeme nejdříve jejich název a datový typ. O řádek níž defujeme pro data LOOKUP_TABLE a pod ní data samotná. Indexy jednotlivých prvků budou indexy uzlů, ke kterým uvedené hodnoty příslušejí.
Pro detailní dokumentaci textové verze VTK odkážeme na [11, str. 1-11].

C.3.2 XML formát

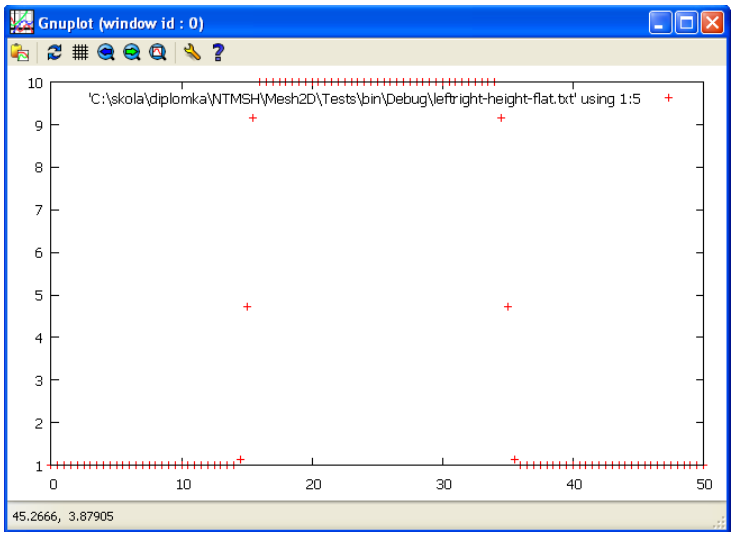
V XML verzi našeho příkladu jsou analogické prvky, jako v textové. Podrobně jsou popsány v [11, str. 11-19].

```
1  <VTKFile type="PolyData" version="0.1" byte_order="LittleEndian">
2    <PolyData>
3      <Piece NumberOfPoints="4" NumberOfVerts="0" NumberOfLines="0" NumberOfStrips="0" NumberOfPolys="1">
4        <PointData Scalars="MyData1">
5          <DataArray type="Float32" Name="MyData1" format="ascii" RangeMin="0" RangeMax="1">
6            0 0.5 0.5 1
7          </DataArray>
8          <DataArray type="Float32" Name="MyData2" format="ascii" RangeMin="0" RangeMax="1">
9            0 0 1 1
10         </DataArray>
11       </PointData>
12       <Points>
13         <DataArray type="Float32" Name="Points" NumberOfComponents="3" format="ascii" RangeMin="0.70710678119" RangeMax="0.70710678119">
14           -0.5 -0.5 0 0.5 -0.5 0 -0.5 0.5 0 0.5 0.5 0
15         </DataArray>
16       </Points>
17       <Polys>
18         <DataArray type="Int32" Name="connectivity" format="ascii" RangeMin="0" RangeMax="3">
19           0 1 3 2
20         </DataArray>
21         <DataArray type="Int32" Name="offsets" format="ascii" RangeMin="4" RangeMax="4">
22           4
23         </DataArray>
24       </Polys>
25     </Piece>
26   </PolyData>
27 </VTKFile>
```

C.4 Data pro program Gnuplot

```
0      25  0      0      1
0.5    25  0.01   0.5    1
1      25  0.02   1      1
1.5    25  0.03   1.5    1
...
14     25  0.28   14     1
14.5   25  0.29   14.5   1.12528367810693
15     25  0.3    15     4.72045484703153
15.5   25  0.31   15.5   9.15426147486154
16     25  0.32   16     10
16.5   25  0.33   16.5   10
...
48.5   25  0.97   48.5   1
49     25  0.98   49     1
49.5   25  0.99   49.5   1
50     25  1      50     1
```

Generujeme graf z prvního a pátého sloupce takovéhoto soubor:
`plot 'C:\tmp\leftright-height-flat.txt' using 1:5`



Obrázek 20: Vizualizace datového souboru programem Gnuplot

D Konfigurační soubor výpočtu

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <configuration name="Testing Configuration">
3    <solver solvername="Default" solvertype="explicit">
4      <flux>VS-flux-Swe</flux>
5      <cfl>  0.9 </cfl>
6      <timestep>  0.01 </timestep>
7      <timelevel> 2 </timelevel>
8      <maxiteration> 1000 </maxiteration>
9      <starttime> 0 </starttime>
10     <stoptime> 0.1 </stoptime>
11     <icquadratureorder> 2</icquadratureorder>
12     <steadystatenorm>1e-4</steadystatenorm>
13   </solver>
14   <solution dimension="3">
15     <component index="0">Heigth</component>
16     <component index="1">XMomentum</component>
17     <component index="2">YMomentum</component>
18   </solution>
19 </configuration>
```

Význam elementů je většinou jasný, až na:

<**flux**> použitý numerický tok

<**timelevel**> počet předchozích iterací, které má iterační metoda použít pro nové řešení

<**steadystatenorm**> norma, při které je dosažen stacionární stav

Odkážeme přímo do XML souboru konfigurace na komentáře, které jsme zde neuvedli.