SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Stavebná fakulta

Riešenie rovnice advekcie pomocou softvéru DUNE

Bakalárska práca

Študijný odbor: Matematicko-počítačové modelovanie

Vedúci bakalárskej práce:
RNDr. Peter Frolkovič, PhD.

Bakalár:
Maroš Bohunčák

Bratislava 2009

Slovenská technická univerzita
v Bratislave
Stavebná fakulta

Riešenie rovnice advekcie pomocou softvéru DUNE.
Solution of advection equation using software DUNE.

Autor práce:	**Maroš Bohunčák**
Ročník/stupeň štúdia:	3. ročník/1. stupeň štúdia
Študijný program:	matematicko-počítačové modelovanie
Vedúci práce:	RNDr. Peter Frolkovič, PhD.
Katedra:	KMaDG

Anotácia:
Práca sa zaoberá návrhom a realizáciou numerickej metódy druhého rádu presnosti na riešenie rovnice advekcie pre dané rýchlostné vektorové pole. Dôraz je kladený na počítačovú realizáciu v modernom softvéri DUNE (Distributed and Unified Numerics Environment), ktorý na základe jednotného rozhrania pre použitie numerických metód na rôznych typoch výpočtových sietí umožňuje ich efektívne použitie od najjednoduchších ako sú rovnomerná, pevná, štruktúrovaná sieť až po komplexné, lokálne zjemnené, v čase premenlivé, neštruktúrované siete. Úlohou študentskej práce je pochopiť základy práce s týmto softvérom, realizovať v ňom novú numerické metódu a preveriť ju na rôznych príkladoch.

Annotation:
This work deals with a proposal and realization of a second order accurate numerical method for the solution of advection equation for a given velocity vector field. The emphasize is given on the implemenntation of such method in modern software tool DUNE (Distributed and Unified Numerics Environment) that offers a single interface for numerical methods based on different types of computational grids to work efficiently with them starting from the simplest uniform, fixed, structured grids and finishing with complex, locally adapted, changing in time, unstructured grids. The aim of this student work is to understand the principles of this software, to realize new numerical method in this environment and to check it on several examples.

Čestné prehásenie

Vyhlasujem, že som bakalársku prácu vypracoval samostatne s použitím uvedenej odbornej literatúry. Bratislava 21. mája 2009

. . . . . .

Vlastnoručný podpis

Poďakovanie

Ďakujem môjmu konzultantovi, RNDr. Petrovi Frolkovičovi, PhD., za cenné rady a podnety, ktoré mi poskytoval počas tvorby tejto práce.

Obsah

1 Úvod

1.1 Čo je vlastne DUNE?

DUNE (*Distributed and Unified Numerics Environment*) je softvérová knižnica napísaná v programovacom jazyku C++ na numerické riešenie parciálnych diferenciálnych rovníc. Sú použité moderné programovacie techniky ako generické programovanie či statický polymorfizmus. Podporuje jednoduchú implementáciu metód ako Metóda konečných prvkov (MKP), Metóda konečných objemov (MKO), alebo Metóda konečných diferencií (MKD). **DUNE** je voľne dostupná pod licenciou GPL. Je možné aj použitie s iným komerčným softvérom. Knižnica **DUNE** je rozdelená do samostatných modulov. Pre aktuálnu verziu 1.2 (z 21. mája 2009) sú dostupné moduly:

- `dune-common` obsahuje základné triedy používané všetkými ostatnými **DUNE** modulmi.
- `dune-grid` je najrozvinutejší modul, ktorý je použitý v tejto práci. Definuje nekonformné, hierarchicky vnorené, paralelné siete a siete s rôznym typom elementov v priestore s *ľubovoľným rozmerom*. Podporuje grafický výstup napríklad vo formáte VTK.
- `dune-istl` - *Iterative Solver Template Library* poskytuje základné triedy riedkych matíc a vektorov a metódy na ich riešenie.
- `dune-grid-howto` - manuál [2] s ukážkovou implementáciou metódy prvého rádu, ktorú sme modifikovali v práci na metódu druhého rádu

1.2 Stiahnutie

Zdrojové kódy **DUNE** je možné stiahnuť z internetovej stránky. V práci sú použité moduly `dune-common`, `dune-grid` a `dune-grid-howto`. Ich posledné stabilné verzie sú dostupné na adrese

`www.dune-project.org/download.html`

Dostupné sú taktiež vývojové verzie.

1.3 Inštalácia

Oficiálne pokyny na inštaláciu sú dostupné na internetovej adrese

`www.dune-project.org/doc/installation-notes.html`

Popíšeme len základnú inštaláciu modulov (verzia 1.2) použitých v tejto práci. Inštrukcie na inštaláciu iných externých modulov a o rôznych variantoch zostavenia inštalácie rozhrania **DUNE** sú popísané na spomenutej stránke. Na zostavenie potrebujeme najlepšie UNIX- ový operačný systém a vyhovujúci C++ kompilátor. V práci bol použitý GNU g++ kompilátor vo verzii 4.3.3 (odporúčaná verzia je $\geq$ 3.4.1).

Stiahnuté tar-balíčky `dune-common`, `dune-grid` a `dune-grid-howto` rozbalíme do spoločného adresára. Prejdeme do tohto adresára, v ktorom sú po rozbalení tri ďalšie adresáre pomenované podľa príslušných modulov a zadáme

```
$ dune-common-1.2/bin/dunecontrol all
```

pre nakonfigurovanie a zostavenie všetkých modulov so základnou konfiguráciou. Môžeme ešte ako superužívateľ zadať

```
# dune-common-1.2/bin/dunecontrol make install
```

aby sa moduly **DUNE** nainštalovali do systému. Na priloženom cd k tejto práci je balíček `M2_dualna_siet.tar.gz`, čo je nami upravený modul `dune-grid-howto`, ktorý obsahuje ukážkové programy z častí 2.9 a 3.2 a program s metódou druhého rádu počítanú na vrcholoch z časti 5.4.

2 Modul DUNE Grid

Popíšeme triedy a metódy modulu `dune-grid`, ktoré sú použité v tejto práci. Pre podrobný popis jednotlivých rozhraní tried viď [5]. K slovenskému názvu vždy v zátvorke uvádzame oficiálny anglický názov použitý v [2].

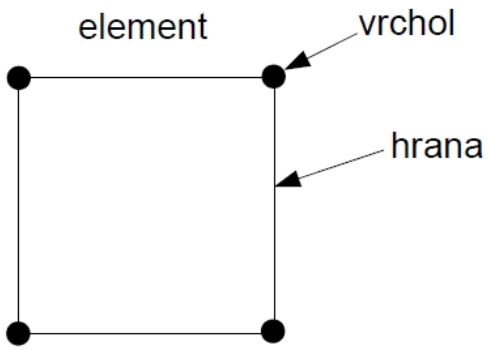
2.1 Sieť (Grid)

Trieda `Dune::Grid`.
Keďže **DUNE** poskytuje nástroje pre metódy sietí (angl. grid-based methods), je modul `dune-grid` jednou z jeho základných častí. Uvedieme niekoľko výrazov, ktoré definujú koncept siete:

Dimenzia (Dimension)
Sieť má pevnú dimenziu d. Je to dimenzia jej referenčných elementov. Používa sa ako parameter šablóny pri vytváraní siete.

Entita (Entity)
Entita je geometrický objekt, ktorý je súčasťou siete. Jej dimenzia je rovná dimenzii siete, alebo je menšia ako dimenzia siete.

Kodimenzia (Codimension)
Každá entita (napr. samotný 2D element, alebo jeho hrany, alebo vrcholy) má svoju kodimenziu c, pre ktorú platí $0 \leq c \leq d$. Platí, že entita kodimenzie c je $d - c$ dimenzionálny objekt.



Obrázok 1: Príklad štvoruholníkového 2D elementu. Element je entita s kodimenziou 0, hrana je entita s kodimenziou 1 a vrchol je entita s kodimenziou 2.

Podentita (Subentity)
Entity majú hierarchickú konštrukciu v tom zmysle, že entity kodimenzie 0 (napr. štvorec

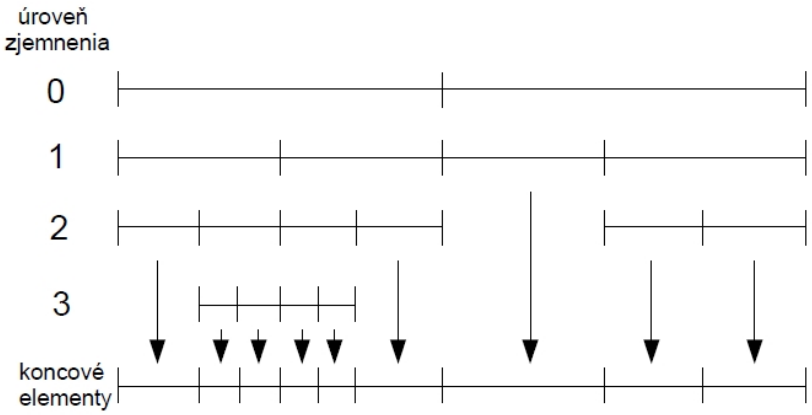
v 2D) sú tvorené entitami s kodimenziou 1 (hrany štvorca), ktoré sú tvorené entitami s kodimenziou 2 (vrcholy štvorca). Platí teda, že maximálna možná kodimenzia nejakej podentity v elemente je rovná dimenzii elementu.

Element (Element)
Element (obr. (1)) je entita kodimenzie 0. Napr. štvorec v 2D, kocka v 3D.

Vrchol (Vertex)
Vrchol (obr. (1)) je entita kodimenzie d (je rovná dimenzii siete).

Priestorová dimenzia (World dimension)
Každá sieť má priestorovú dimenziu w. Je to dimenzia priestoru, v ktorom je sieť umiestnená, alebo inak povedané je to počet súradníc potrebný na popis pozície ľubovoľného vrcholu v sieti v priestore, v ktorom je sieť umiestnená. Platí, že $w \geq d$. Spolu s dimenziou siete sa používa ako parameter šablóny pri vytváraní siete.

Koncová sieť (Leaf grid)
Keďže pri viacnásobnom zjemňovaní siete dostávame tzv. stromovú štruktúru zjemňovania, konce tohto stromu tvoria tzv. koncovú sieť (obr. (2)). Sú to teda všetky koncové elementy (listy) siete, ktoré už nie sú delené na menšie elementy (nie sú zjemnené). Hovoríme, že je to sieť s najjemnejším rozlíšením.



Obrázok 2: Koncové elementy (listy) siete.

2.2 Zjemnenie siete (Grid refinement)

Sieť môže byť zjemnená globálne, alebo lokálne len v takzvanej zjemňovacej fáze. Keďže koncept siete je v **DUNE** založený na *view-only* princípe, mimo tejto fázy sieť nemôže byť nijakým spôsobom modifikovaná.

2.3 Pohľady na sieť (Grid views)

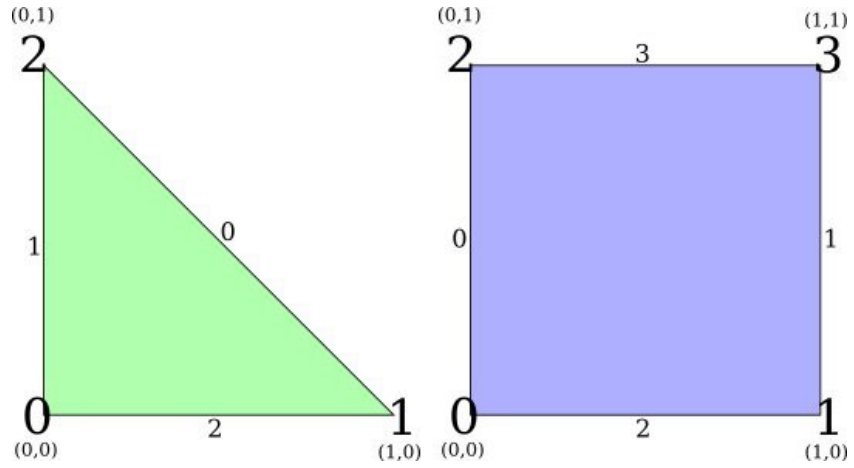
Trieda `Dune::GridView`.
Sieť poskytuje dva pohľady: pohľad cez úrovne (Level view), alebo pohľad cez koncové elementy (Leaf view). Pohľad cez úrovne používame, keď pracujeme s lokálne zjemnenými sieťami. V tejto práci nebudeme sieť lokálne zjemňovať a budeme používať pohľad cez koncové elementy.

2.4 Entity (Entities)

Trieda `Dune::Entity`.
Pre jednotlivé entity siete všetkých kodimenzií platí, že ich nemôžeme v sieti modifikovať ani vytvárať, ani mazať. Môžeme z nich informácie *len čítať*. Každá entita nesie o sebe *topologickú* (či sa jedná o trojuholník, štvoruholník atď.) a *geometrickú* (pozícia entity v sieti) informáciu. Entita je definovaná *referenčným elementom* a *transformáciou* z referenčného elementu do globálnych súradníc.

2.5 Referenčný element (Reference element)

Trieda `Dune::ReferenceElement`.
Každá entita siete je výsledkom zobrazenia z referenčného elementu do globálnych súradníc. Topológia referenčného elementu je popísaná len raz (t.j. všetky elementy v sieti, napr. štvoruholníky, majú jeden spoločný referenčný element). Referenčný element popisuje, ako je daná entita zostavená z iných entít vyššej kodimenzie a akých typov sú tieto podentity. V tejto práci sa obmedzíme na 2D elementy.



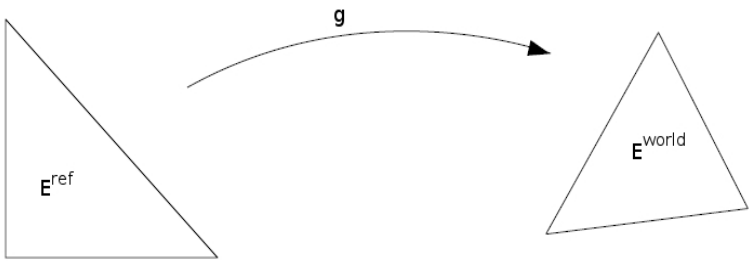
Obrázok 3: Príklad referenčného elementu pre trojuholníkový a štvoruholníkový element s očíslovanými vrcholmi a hranami.

2.6 Geometria (Geometry)

Trieda `Dune::Geometry`.
Geometria definuje zobrazenie (transformáciu) [2]

$$g : D \rightarrow W \tag{1}$$

kde $D \subseteq \mathbb{R}^{mydim}$ a $W \subseteq \mathbb{R}^{cdim}$. Oblasť D je z množiny referenčných elementov $E^{ref} \in D$. Vo všeobecnosti platí $mydim \leq cdim$. V práci budeme uvažovať $mydim = cdim = 2$. Pomocou metód triedy budeme získavať informácie o lokálnych a globálnych súradniciach entity, jej veľkosť, alebo inverzný Jakobián zobrazenia.



Obrázok 4: Transformácia referenčného trojuholníkového elementu.

2.7 Priesečníky (Intersections)

Priesečníkmi sú vo všeobecnosti pomenované hrany (v 2D prípade), alebo plochy (v 3D prípade), ktorými sa susedné elementy dotýkajú. Priesečník môže mať element s iným (susedným) elementom, alebo s okrajom výpočtovej siete. Cez priesečníky elementu môžeme iterovať pomocou IntersectionIterator. Tento iterátor zároveň poskytuje informácie o topológii a geometrii priesečníka.

2.8 Iterátor a ukazovateľ (Iterator, Entity Pointer)

Triedy `Dune::EntityPointer`, `Dune::Iterator`, `Dune::IntersectionIterator`.
Na jednotlivé entity siete pristupujeme pomocou iterátorov a ukazovateľov (obr. (5)). V práci sme použili:

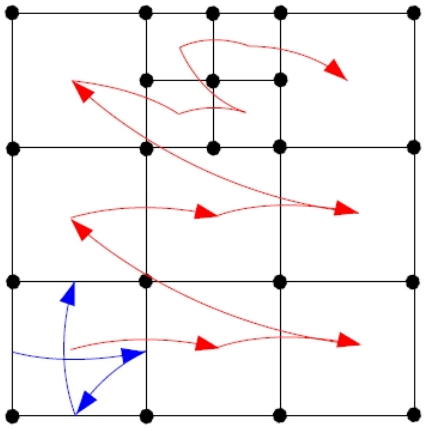
Entity Pointer
Ukazovateľ na entitu.

Iterator

Iterator je ukazovateľ na entitu, ktorý môže byť zvyšovaný, aby ukazoval na ďalšiu entitu v poradí - v prípade pohľadu cez listy, alebo aby ukazoval na ďalšiu entitu v poradí *na danej úrovni zjemnenia* - v prípade pohľadu cez úrovne.

IntersectionIterator

Iterátor cez priesečníky elementu so susedným elementom, alebo okrajom výpočtovej siete. Viď nasledujúcu časť.



Obrázok 5: Príklad iterátora cez koncové elementy - červená a iterátora cez priesečníky v jednom elemente - modrá.

2.9 Ukážka - prechádzka po sieti s DUNE

Na záver tejto časti uvádzame základný algoritmus, ktorý prejde najprv cez každý koncový element 2D siete a potom cez každý vrchol siete.

```
1  // C/C++ hlavickove subory
2  #include<iostream>
3
4  // DUNE hlavickove subory
5  #include"config.h"   // obsahuje informacie o dostupnych typoch sieti
6  #include<dune/grid/sgrid.hh> // pouzijeme siet typu SGrid
7
8  template<class G>
9  void traversal (G& grid)
10 {
11   // zistime dimenziu siete
12   const int dim = G::dimension;
13
14   // typ suradnic
15   typedef typename G::ctype ct;
16
17   // prechadzka po koncovej sieti
18   std::cout << "*** Prechadzam koncove elementy" << std::endl;
19
```

```
20    // typ pohladu
21    typedef typename G :: LeafGridView LeafGridView;
22
23    // pohlad
24    LeafGridView leafView = grid.leafView();
25
26    // iterator cez elementy (entity s kodimenziou 0)
27    typedef typename LeafGridView::template Codim<0>::Iterator ElementLeafIterator;
28
29    int count = 0;
30    for (ElementLeafIterator it = leafView.template begin<0>();
31         it!=leafView.template end<0>(); ++it)
32          {
33            // geometria elementu
34            Dune::GeometryType gt = it->type();
35
36            std::cout << "prechadzam " << gt
37                  << " so zaciatocnym vrcholom " << it->geometry().corner(0)
38                                << std::endl;
39            count++;
40          }
41
42    std::cout << "spolu je tu " << count << " koncovych elementov" << std::endl;
43
44    // teraz prejdeme po vrcholoch siete
45    std::cout << std::endl;
46    std::cout << "*** Prechadzam entity s kodimenziou " << dim << std::endl;
47
48    // iterator cez vrcholy (entity s kodimenziou dim)
49    typedef typename LeafGridView::template Codim<dim>::Iterator VertexLeafIterator;
50
51    // iterujeme cez vrcholy
52    count = 0;
53    for (VertexLeafIterator it = leafView.template begin<dim>();
54         it!=leafView.template end<dim>(); ++it)
55          {
56            Dune::GeometryType gt = it->type();
57            std::cout << "prechadzam " << gt
58                  << " na " << it->geometry().corner(0)
59                                << std::endl;
60            count++;
61          }
62    std::cout << "Spolu je tu " << count << " vrcholov"
63                          << std::endl;
64  }
65
66  // hlavny program
67  int main(int argc, char **argv)
68  {
69
70    // try/catch blok, DUNE vrati chybu ak sa nejaka vyskytne
71    try {
72      // vyroba siete
73      const int dim=2;
74      // typ siete
75      typedef Dune::SGrid<dim,dim> GridType;
76      // pocet elementov siete
77      Dune::FieldVector<int,dim> N(1);
78      // lavy dolny roh (-1,-1)
79      Dune::FieldVector<GridType::ctype,dim> L(-1.0);
80      // pravy horny roh (1,1)
81      Dune::FieldVector<GridType::ctype,dim> H(1.0);
82
83      GridType grid(N,L,H);
84
85      // jedenkrat zjemnime siet
86      grid.globalRefine(1);
87
88      // funkcia prechadzky
```

```
89      traversal(grid);
90    }
91    catch (std::exception & e) {
92      std::cout << "STL ERROR: " << e.what() << std::endl;
93      return 1;
94    }
95    catch (Dune::Exception & e) {
96      std::cout << "DUNE ERROR: " << e.what() << std::endl;
97      return 1;
98    }
99    catch (...) {
100     std::cout << "Unknown ERROR" << std::endl;
101     return 1;
102   }
103
104   return 0;
105 }
```

Výstup vyzerá takto:

```
*** Prechadzam koncove elementy
prechadzam (cube, 2) so zaciatocnym vrcholom -1 -1
prechadzam (cube, 2) so zaciatocnym vrcholom 0 -1
prechadzam (cube, 2) so zaciatocnym vrcholom -1 0
prechadzam (cube, 2) so zaciatocnym vrcholom 0 0
spolu je tu 4 koncovych elementov

*** Prechadzam entity s kodimenziou 2
prechadzam (cube, 0) na -1 -1
prechadzam (cube, 0) na 0 -1
prechadzam (cube, 0) na 1 -1
prechadzam (cube, 0) na -1 0
prechadzam (cube, 0) na 0 0
prechadzam (cube, 0) na 1 0
prechadzam (cube, 0) na -1 1
prechadzam (cube, 0) na 0 1
prechadzam (cube, 0) na 1 1
spolu je tu 9 vrcholov
```

`(cube,2)`, resp. `(cube,0)` je typ geometrie a kodimenzia entity. T.j. entita s typom geometrie štvoruholník a s kodimenziou 2 (samotný element), resp. entita s typom geometrie štvoruholník a s kodimenziou 0 (vrchol). Pre ostatné typy geometrie viď [5].

3 Práca s dátami na sieti

V aplikáciách potrebujeme priradiť užívateľské dáta konkrétnym entitám siete. V tejto časti popíšeme ako dáta možno priradiť sieti.

3.1 Zobrazovače (Mappers)

Jednou zo základných vlastností **DUNE** grid rozhrania je oddelenie siete od dát užívateľa. Inak povedané, *sieť ako objekt nepotrebuje žiadne informácie o užívateľských dátach.* Dáta užívateľa sú kvôli úspornosti ukladané do jednorozmerných polí a pristupuje sa k nim prostredníctvom indexov, ktoré sú číslované od nuly.

Predpokladajme, že množina všetkých entít na sieti je E a $E' \subseteq E$ je podmnožina tých entít, pre ktoré chceme dáta uložiť. Napríklad to môžu byť všetky vrcholy siete. Potom prístup z týchto entít k dátam tvoria dva kroky: Takzvaný zobrazovač realizuje zobrazenie

$$m : E' \to I_{E'} \tag{2}$$

kde $I_{E'} = \{0, \ldots, |E'| - 1\} \subset \mathbb{N}$ je množina indexov (postupnosť so začiatkom v nule) priradená množine entít. Užívateľské dáta $D(E') = \{d_e | e \in E'\}$ sú uložené v poli, čo je ďalšie zobrazenie

$$\alpha : I_{E'} \to D(E'). \tag{3}$$

Ak teda chceme priradiť dáta $d_e \in D(E')$ entite $e \in E'$ treba uskutočniť dve zobrazenia:

$$d_e = \alpha(m(e)). \tag{4}$$

DUNE implementuje dva typy zobrazovačov, ktoré sa líšia vo funkcionalite a náročnosti (čo sa týka kapacity a času behu):

Indexové zobrazovače
Trieda `Dune::LeafMultipleCodimMultipleGeomTypeMapper`.
Indexový zobrazovač je alokovaný pre sieť a má zmysel s ním pracovať, pokiaľ sieť nie je zmenená (zjemňovaná). Základom implementácie je trieda `Dune::IndexSet`, ktorá realizuje zobrazenie (2). V práci sú použité len indexové zobrazovače.

Id zobrazovače
Id zobrazovače sa používajú ak sa sieť počas výpočtov zjemní. V tejto práci ich nebudeme používať.

3.2 Ukážka - priradenie dát elementom

Tento krátky program priradí každému elementu siete hodnotu funkcie `f` v daných súradniciach stredu elementu:

```
1  #include<dune/grid/common/referenceelements.hh>
2  #include<dune/grid/common/mcmgmapper.hh>
3  #include<dune/grid/io/file/vtk/vtkwriter.hh>
4
5  // funkcia na vypocet hodnot
6  template<int dimworld, class ct>
7  double f (const Dune::FieldVector<ct,dimworld>& x)
8  {
9    return sqrt(pow(x[0]-.5,2)+pow(x[1]-.75,2))-.15;
10 }
11
12 // parameter pre mapper
13 template<int dim>
14 struct P0Layout
15 {
16   bool contains (Dune::GeometryType gt)
17   {
18         if (gt.dim()==dim) return true;
19         return false;
20   }
21 };
22
23 // priradenie dat elementom
24 template<class G, class F>
25 void elementdata (const G& grid, const F& f)
26 {
27   // zistenie dimenzii, typu suradnic, nastavenie pohladu a typu iteratora
28   const int dim = G::dimension;
29   const int dimworld = G::dimensionworld;
30   typedef typename G::ctype ct;
31   typedef typename G::LeafGridView GridView;
32   typedef typename GridView::template Codim<0>::Iterator ElementLeafIterator;
33
34   // pohlad na koncovu siet
35   GridView gridView = grid.leafView();
36
37   // mapper pre entity kodimenzie 0 na koncovej sieti
38   Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P0Layout> mapper(grid);
39
40   // vektor na ulozenie dat
41   std::vector<double> c(mapper.size());
42
43   // iterujeme cez entity kodimenzie 0 (elementy)
44   for (ElementLeafIterator it = gridView.template begin<0>();
45         it!=gridView.template end<0>(); ++it)
46       {
47         // typ geometrie elementu
48         Dune::GeometryType gt = it->type();
49
50         // stred v referencnom elemente
51         const Dune::FieldVector<ct,dim>&
52               local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
53
54         // stred v globalnych suradniciach
55         Dune::FieldVector<ct,dimworld> global = it->geometry().global(local);
56
57         // vypocet funkcie a ulozenie hodnoty
58         c[mapper.map(*it)] = f(global);
59       }
60
61   // generovanie VTK suboru
```

```
62    Dune::VTKWriter<typename G::LeafGridView> vtkwriter(gridView);
63    vtkwriter.addCellData(c,"data");
64    vtkwriter.write("elementdata",Dune::VTKOptions::binaryappended);
65  }
```

Indexový zobrazovač je vytvorený na riadku 38. Entity v podmnožine E' sú *všetky* koncové entity a môžu byť ďalej vyberané z tejto množiny podľa *dimenzie, kodimenzie a typu geometrie*. Kvôli tomu je jeden parameter pri vytváraní zobrazovača štruktúra `P0layout`, ktorej metóda `contains` vráti `true`, ak má entita nami požadovanú dimenziu, kodimenziu a typ geometrie (riadky 12-21). Na riadku 40 je alokovaný vektor pre uloženie dát. Cyklus na riadkoch 44-59 iteruje cez elementy siete a každom elemente sú vypočítané súradnice jeho stredu, ktoré sú použité ako premenné vo funkcii `f`. Vypočítaná hodnota je potom uložená na príslušné miesto v poli `c`. Zobrazenie m je realizované volaním `mapper.map(*it)` kde `*it` je entita, ktorú spracuje zobrazovač. Zvyšné riadky kódu generujú VTK súbor, ktorý možno zobraziť napríklad v softvéri ParaView. Ak by sme dáta ukladali pre vrcholy siete, stačí zmeniť riadok 63 na `vtkwriter.addVertexData(c,"data");`

4 Metóda konečných objemov

Popis metódy konečných objemov a numerickej metódy prvého rádu sme prebrali z [2]. Budeme riešiť lineárnu hyperbolickú parciálnu diferenciálnu rovnicu, alebo tzv. rovnicu advekcie

$$\frac{\partial c}{\partial t} + \nabla \cdot (uc) = 0 \quad \text{na} \quad \Omega \times T \tag{5}$$

kde $\Omega \subset \mathbb{R}$ je výpočtová oblasť, $T = (0, t_{end})$ je časový interval, $c : \Omega \times T \to \mathbb{R}$ je neznáma funkcia a $u : \Omega \times T \to \mathbb{R}^d$ je dané rýchlostné pole. Divergencia rýchlostného poľa nech je nulová. Pre rovnicu máme definovanú počiatočnú podmienku

$$c(x, 0) = c_0(x) \quad x \in \Omega \tag{6}$$

a okrajovú podmienku

$$c(x, t) = b(x, t) \quad t > 0, x \in \Gamma_{in}(t) = \{y \in \partial\Omega \quad | \quad u(y, t) \cdot \nu(y) < 0\}. \tag{7}$$

$\nu(y)$ je jednotkový vektor vonkajšej normály v bode $y \in \partial\Omega$ a $\Gamma_{in}(t)$ je prítoková hranica v čase t.

Spojitú diferenciálnu rovnicu (5) aproximujeme diskrétnymi rovnicami pomocou metódy konečných objemov. Výpočtová sieť pozostáva z elementov ω_i a časový interval T je diskretizovaný na kroky $0 = t_0, t_1, \ldots, t_n, t_{n+1}, \ldots, t_N = t_{end}$. Rovnicu (5) prepíšeme do integrálneho tvaru, t.j. integrujeme cez každý element výpočtovej siete ω_i a časový interval (t_n, t_{n+1}):

$$\int_{\omega_i} \int_{t_n}^{t_{n+1}} \frac{\partial c}{\partial t} \, dt \, dx + \int_{\omega_i} \int_{t_n}^{t_{n+1}} \nabla \cdot (uc) \, dt \, dx = 0 \quad \forall i. \tag{8}$$

Po čiastočnej integrácii dostávame

$$\int_{\omega_i} c(x, t_{n+1}) \, dx - \int_{\omega_i} c(x, t_n) \, dx + \int_{t_n}^{t_{n+1}} \int_{\partial\omega_i} cu \cdot \nu \, ds \, dt = 0 \quad \forall i \tag{9}$$

Hranicu elementu $\partial\omega_i$ rozdelíme na časti γ_{ij}, čo je buď prienik s iným (susedným) elementom $\partial\omega_i \cap \partial\omega_j$, alebo prienik s hranicou oblasti $\partial\omega_i \cap \partial\Omega$.
Aproximácia integrálov v (8) vedie k nasledujúcej rovnici pre neznáme hodnoty C_i^{n+1} elementov v čase t_{n+1}

$$C_i^{n+1}|\omega_i| - C_i^n|\omega_i| + \sum_{\gamma_{ij}} F_{ij}^{n+\frac{1}{2}} \, |\gamma_{ij}| \triangle t^n = 0 \quad \forall i, \tag{10}$$

kde $\triangle t^n = t_{n+1} - t_n$, $u_{ij}^{n+\frac{1}{2}}$ je rýchlosť na hranici elementu γ_{ij} v jej strednom bode $\mathbf{x}_{ij} \in \gamma_{ij}$ v čase $t_{n+\frac{1}{2}} = t_n + \frac{1}{2}\triangle t^n$, ν_{ij} je jednotkový vektor vonkajšej normály časti hranice γ_{ij} a funkcia toku je definovaná ako

$$F_{ij}^{n+\frac{1}{2}} = \begin{cases} b(\mathbf{x}_{ij},\, t_{n+\frac{1}{2}}) & \mathbf{x}_{ij} \in \Gamma_{in}(t_{n+\frac{1}{2}}) \\ C_{ji}^{n+\frac{1}{2}}\, u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij} & u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij} < 0 \\ C_{ij}^{n+\frac{1}{2}}\, u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij} & u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij} \geq 0 \end{cases} \tag{11}$$

Hodnota $b(\mathbf{x}_{ij})$ je daná okrajovou podmienkou na prítokovej časti hranice elementu γ_{ij}. Hodnoty $C_{ji}^{n+\frac{1}{2}}$ upresníme neskôr. Ak zvolíme $C_{ji}^{n+\frac{1}{2}} = b(\mathbf{x}_{ij})$ na $\mathbf{x}_{ij} \in \Gamma_{in}(t)$, môžeme tvar (11) prepísať na

$$F_{ij}^{n+\frac{1}{2}} = C_{ij}^{n+\frac{1}{2}}\, \max(0, u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}) - C_{ji}^{n+\frac{1}{2}}\, \max(0, -u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}). \tag{12}$$

4.1 Numerická metóda prvého rádu

4.1.1 Metóda na elementoch

Aproximujme $c(x,t)$ funkciou $C(x,t) = C_i^n$, $x \in \omega_i$, $t \in [t_n, t_{n+1})$, pričom C_i^n je hodnota na elemente ω_i v čase t_n. Pri metóde prvého rádu zvolíme $C_{ij}^{n+\frac{1}{2}} = C_i^n$ a funkcia toku má tvar

$$F_{ij}^{n+\frac{1}{2}} = C_i^n\, \max(0, u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}) - C_j^n\, \max(0, -u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}). \tag{13}$$

Z (10) po rozpísaní a vyjadrení C_i^{n+1} dostávame

$$C_i^{n+1} = C_i^n - C_i^n \triangle t^n \sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}) + \triangle t^n \sum_{\gamma_{ij}} C_j^n \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, -u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij})\, \forall i. \tag{14}$$

Schéma je stabilná, ak je splnené

$$\forall i : 1 - \triangle t^n \sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}) \geq 0 \Leftrightarrow \triangle t^n \leq \min_i \left(\sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}) \right)^{-1} \tag{15}$$

Prepíšeme (14) na tvar

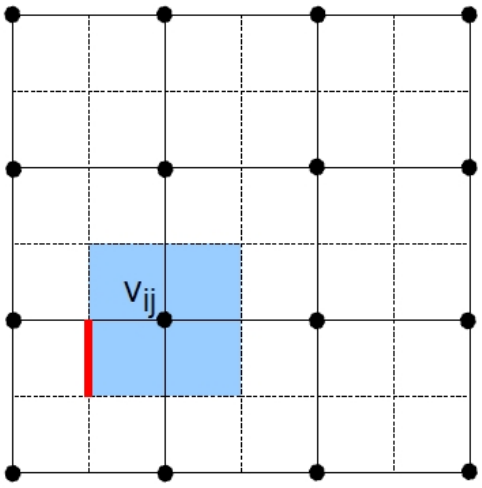
$$C_i^{n+1} = C_i^n - \triangle t^n \underbrace{\sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|\omega_i|} (C_i^n \max(0, u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}) + C_j^n \max(0, -u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}))}_{\delta_i} \quad \forall i \tag{16}$$

4.1.2 Metóda na vrcholoch

Pri metóde počítanej na vrcholoch, ktorú sme navrhli, využijeme tzv. duálnu sieť (obr. (6)). Tok počítame cez príslušný element ω_i *duálnej siete*. Výpočet hodnôt funkcie C_i^{n+1} je rovnaký ako pri metóde na elementoch:

$$C_i^{n+1} = C_i^n - C_i^n \triangle t^n \sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}) + \triangle t^n \sum_{\gamma_{ij}} C_j^n \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, -u_{ij}^{n+\frac{1}{2}} \cdot \nu_{ij}) \, \forall i. \tag{17}$$

kde γ_{ij} je teraz segment duálnej siete.



Obrázok 6: Duálna sieť (čiarkovaná). Tok počítame cez príslušný element duálnej siete (modrý), ktorého hranica pozostáva zo segmentov (na obrázku je vyznačený jeden červenou farbou) na duálnej sieti.

4.2 Numerická metóda druhého rádu

Pri metóde prvého rádu sme funkciu c aproximovali konštantnou funkciou. Pri metóde druhého rádu [1] použijeme

$$C^n(\mathbf{x}) = C_i^n + \nabla C_i^n \cdot (\mathbf{x} - \mathbf{x}_i) \tag{18}$$

kde $C^n(\mathbf{x})$ je lineárna funkcia a

$$C_{ij}^{n+\frac{1}{2}} = C^n(\mathbf{x}_{ij}) - \frac{\triangle t^n}{2} \nabla C_i^n \cdot u_i^{n+1/2}, \tag{19}$$

kde $\mathbf{x}_{ij}$ sú súradnice stredu priesečníka dvoch susedných elementov (časť hranice, ktorou sa elementy dotýkajú). Ďalej upresníme výpočet gradientu.

4.2.1 Gradient v metóde na elementoch

Gradient pre vnútorný element siete ω_i počítame ako

$$\nabla C_i = \left(\frac{C_{i_{x+1}} - C_{i_{x-1}}}{2h}, \frac{C_{i_{y+1}} - C_{i_{y-1}}}{2h} \right) \tag{20}$$

a pre hraničný element na (napríklad ľavom) okraji oblasti

$$\nabla C_i = \left(\frac{C_{i_{x+1}} - C_i}{h}, \frac{C_{i_{y+1}} - C_{i_{y-1}}}{2h} \right) \tag{21}$$

kde $C_{i_{x+1}}$(resp.$C_{i_{x-1}}$) je hodnota na nasledujúcom (resp. predchádzajúcom) elemente v smere osi x a analogicky aj pre y-ový smer.

4.2.2 Gradient v metóde na vrcholoch

Pri metóde na vrcholoch budeme počítať gradient najprv v referenčnom elemente (lokálny gradient), ktorý vynásobíme inverzným Jakobiánom $J_g^{-T}(x)$ zobrazenia $g(x)$ (1), viď časť 5.4. Teda, ak chceme počítať gradient z funkcie $f : W \to \mathbb{R}$ v mieste $y = g(x)$, $x \in D$ a ak $f^{ref}(x) = f(g(x))$, potom máme:

$$\nabla f(g(x)) = J_g^{-T}(x) \nabla f^{ref}(x) \tag{22}$$

4.3 Výpočet chyby

Na demonštráciu konvergencie oboch metód použijeme jednoduchý výpočet pre tzv. L1 chybu

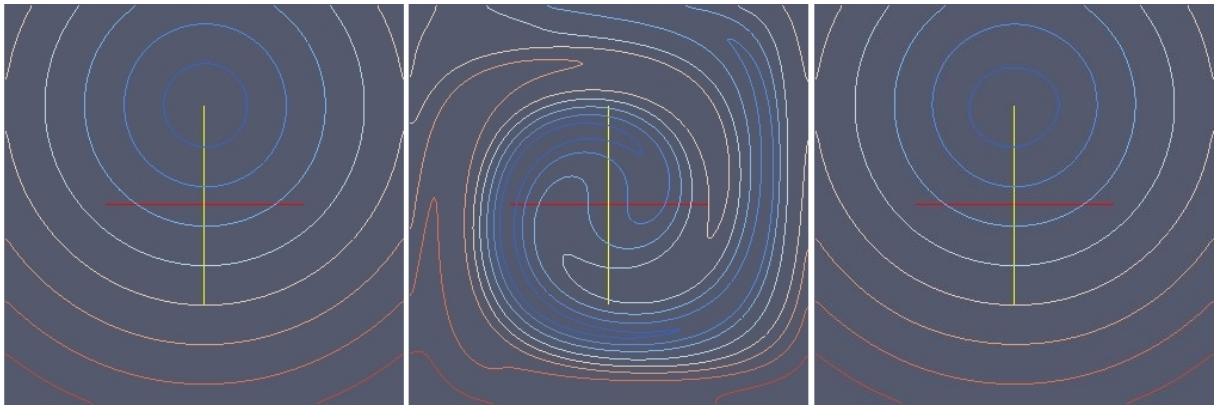
$$E = h^2 \sum_{\omega_i} |c_i^n - cp_i^n| \tag{23}$$

kde cp je presné riešenie v $x_i \in \omega_i$.

5 Implementácia v DUNE

Obe metódy predvedieme na takzvanom single vortex príklade, ktorý je často využívaný na testovanie numerických metód. Úlohu riešime na jednotkovom štvorci $[0,1] \times [0,1]$. Počiatočná podmienka je funkcia $\phi(0,x,y) = \sqrt{(x-0.5)^2 + (y-0.75)^2} - 0.15 = 0$. Časovo závislé rýchlostné pole je dané ako $\mathbf{V}(x,y,t) = cos(\frac{\pi t}{8})(u(x,y), v(x,y))$, (platí $\nabla \cdot \mathbf{V}(x,y,t) = 0$), pričom: $u(x,y) = -sin^2(\pi x)sin(\pi y)cos(\pi y)$, $v(x,y) = sin^2(\pi y)sin(\pi x)cos(\pi x)$.

Počiatočná funkcia, ktorej kontúry majú na začiatku tvar kruhu, sa postupne v čase deformujú. V čase $t = 4$, keď je dosiahnutá maximálna deformácia, sa začína tvar funkcie vracať do pôvodného tvaru, ktorý dosiahne v čase $t = 8$, viď obr. 7.



Obrázok 7: Kontúry numerického riešenia príkladu single vortex v časoch $t = 0$, $t = 4$ a $t = 8$.

5.1 Metóda prvého rádu počítaná na elementoch

Ako prvé si zadefinujeme počiatočné podmienky, okrajové podmienky a rýchlostné pole.

```
1  #define pi 3.14159265358979323846264338327950288419716939937510
2
3  // pociatocna podmienka c0
4  template<int dimworld, class ct>
5  double c0 (const Dune::FieldVector<ct,dimworld>& x)
6  {
7    return sqrt(pow(x[0]-.5,2)+pow(x[1]-.75,2))-.15;
8  }
9
10 // casovo zavisle rychlostne pole
11 template<int dimworld, class ct>
12 Dune::FieldVector<double,dimworld> u (const Dune::FieldVector<ct,dimworld>& x, double t)
13 {
14   Dune::FieldVector<double,dimworld> r;
15   r[0] = -cos(pi*t/8.0)*pow(sin(pi*x[0]),2)*sin(pi*x[1])*cos(pi*x[1]);
16   r[1] =  cos(pi*t/8.0)*pow(sin(pi*x[1]),2)*sin(pi*x[0])*cos(pi*x[0]);
17   return r;
18 }
19
```

```
20 // okrajova podmienka na vtokovej casti hranice
21 template<int dimworld, class ct>
22 double b (const Dune::FieldVector<ct,dimworld>& x, double t)
23 {
24   return c0(x);
25 }
```

Samotný výpočet (16) implementuje funkcia `evolve` s časovým krokom pre ktorý platí (15).

```
1  #include<dune/grid/common/referenceelements.hh>
2
3  template<class G, class M, class V, class Z>
4  void evolve (const G& grid, const M& mapper, V& c, double t, double& dt, V& cp, Z& gr)
5  {
6    // zistenie dimenzie mriezky
7    const int dim = G::dimension;
8    const int dimworld = G::dimensionworld;
9
10   // typ suradnic pouzitych v mriezke
11   typedef typename G::ctype ct;
12
13   // typ pohladu
14   typedef typename G::LeafGridView GridView;
15
16   // typ iteratora cez elementy
17   typedef typename GridView::template Codim<0>::Iterator LeafIterator;
18
19   // typ iteratora cez priesecniky
20   typedef typename GridView::IntersectionIterator IntersectionIterator;
21
22   // typ ukazovatela na entitu kodimenzie 0
23   typedef typename G::template Codim<0>::EntityPointer EntityPointer;
24
25   // pripocitavaci vektor
26   V update(c.size());
27   for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;
28
29   // cyklus na vypocet pripocitavacieho vektora
30   LeafIterator endit = gridView.template end<0>();
31   for (LeafIterator it = gridView.template begin<0>(); it!=endit; ++it)
32         {
33           // typ geometrie
34           Dune::GeometryType gt = it->type();
35
36           // stred elementu v referencnom elemente
37           const Dune::FieldVector<ct,dim>&
38                 local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
39
40           // stred elementu v globalnych suradniciach
41           Dune::FieldVector<ct,dimworld>
42                 global = it->geometry().global(local);
43
44           // velkost elementu
45           double volume = it->geometry().integrationElement(local)
46                 *Dune::ReferenceElements<ct,dim>::general(gt).volume();
47
48           // index elementu
49           int indexi = mapper.map(*it);
50
51           // cyklus cez vsetky hrany elementu
52           IntersectionIterator isend = gridView.iend(*it);
53           for (IntersectionIterator is = gridView.ibegin(*it); is!=isend; ++is)
54                 {
55                   // typ geometrie hrany
56                   Dune::GeometryType gtf = is.intersectionSelfLocal().type();
57
58                   // stred hrany v referencnom elemente
59                   const Dune::FieldVector<ct,dim-1>&
```

```
60                          facelocal =
61                   Dune::ReferenceElements<ct,dim-1>::general(gtf).position(0,0);
62
63                  // vypocet normaloveho vektora
64                  Dune::FieldVector<ct,dimworld> integrationOuterNormal
65                          = is.integrationOuterNormal(facelocal);
66                  integrationOuterNormal
67                          *= Dune::ReferenceElements<ct,dim-1>::general(gtf).volume();
68
69                  // stred hrany v globalnych suradniciach
70                  Dune::FieldVector<ct,dimworld>
71                          faceglobal = is.intersectionGlobal().global(facelocal);
72
73                  // rychlost v strede hrany
74                  Dune::FieldVector<double,dim> velocity = u(faceglobal,t);
75
76                  // vypocet suciny a normaloveho vektora
77                  double factor = velocity * integrationOuterNormal/volume;
78
79                  // rozlisovanie typu suseda
80                  if (is.neighbor())
81                  {
82                    // pristup na suseda
83                    EntityPointer outside = is.outside();
84                    int indexj = mapper.map(*outside);
85
86                    // vypocet tokov
87                    if ( it->level()>outside->level() ||
88                         (it->level()==outside->level() && indexi<indexj) )
89                    {
90                       // sucin rychlosti a normaloveho vektora v susednom elemente
91                       Dune::GeometryType nbgt = outside->type();
92                       const Dune::FieldVector<ct,dim>&
93                    nblocal = Dune::ReferenceElements<ct,dim>::general(nbgt).position(0,0);
94                       double nbvolume = outside->geometry().integrationElement(nblocal)
95                        * Dune::ReferenceElements<ct,dim>::general(nbgt).volume();
96                       double nbfactor = velocity*integrationOuterNormal/nbvolume;
97
98                        if (factor<0) // tok dovnutra
99                        {
100                         update[indexi] -= c[indexj] * factor;
101                         update[indexj] += c[indexj] * nbfactor;
102                       }
103
104                       else // tok von
105                       {
106                         update[indexi] -= c[indexi] * factor;
107                         update[indexj] += c[indexi] * nbfactor;
108                       }
109                     }
110                  }
111
112                // ak je vonkajsia hranica
113                if (is.boundary())
114                {
115                  if (factor<0) // tok dovnutra, aplikacia okrajovej podmienky
116                    update[indexi] -= b(faceglobal,t) * factor;
117
118                  else // tok von
119                  {
120                    update[indexi] -= c[indexi] * factor;
121                  }
122                }
123         } // koniec cyklu cez hranice elementu
124
125     // presne riesenie
126     cp[indexi] = b(global,t+dt);
127
128   } // koniec cyklu cez mriezku
```

```
129
130     // nove riesenie
131     for (unsigned int i=0; i<c.size(); ++i)
132           c[i] += dt*update[i];
133
134     return;
135   }
```

Riadky 30-128 obsahujú cyklus cez všetky elementy siete, v ktorom sa počíta vektor δ_i. Ten je alokovaný na riadku 26, kde predpokladáme že `V` je objekt s kopírovacím konštruktorom a metódou `size`.

Výpočet tokov je implementovaný v riadkoch 80-122. Pomocou `IntersectionIterator` pristupujeme na časti hranice γ_{ij} elementu ω_i. Ak γ_{ij} je priesečník so susedným elementom ω_i metóda iterátora `neighbor()` vráti true (riadok 80), alebo γ_{ij} je priesečník s hranicou výpočtovej oblasti ak metóda `boundary()` vráti true (riadok 113).

A hlavný program:

```
1   // C++ hlavickove subory
2   #include"config.h"
3   #include<iostream>
4   #include<fstream>
5   #include<vector>
6
7   // dune hlavickove subory
8   #include<dune/grid/sgrid.hh> // typ siete
9   #include"vtkout.hh" // na zapisovanie vysledkov
10  #include"transportproblem2.hh" // definicie zac. a okr. podmienok a rych. pola
11  #include"initialize.hh" // inicializacia pociatocnej podmienky
12  #include"evolve.hh" // vypocet MKO
13
14  template<class G>
15  void timeloop (const G& grid, double tend)
16  {
17    // vytvorenie mappera
18    Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P0Layout>
19          mapper(grid);
20
21    // vektor na numericke riesenie, presne riesenie a gradient
22    std::vector<double> c(mapper.size());
23    std::vector<double> cp(mapper.size());
24    std::vector<grad> gr(mapper.size());
25
26    // vypocet hodnot z pociatocnej podmienky
27    initialize(grid,mapper,c,cp,gr);
28
29    // zapis pociatocnych hodnot
30    vtkout(grid,c,"concentration",0);
31
32    // casove kroky
33    double t=0,dt;
34    int k=0;
35    const double saveInterval = 0.1;
36    double saveStep = 0.1;
37    int counter = 0;
38
39    // hlavny cyklus vypoctu hodnot v novom case
40    while (t<tend)
41          {
42            // pocitadlo krokov
43            ++k;
```

```
44
45            // numericky vypocet
46            evolve(grid,mapper,c,t,dt,cp);
47
48            // casovy krok
49            t += dt;
50
51            // zapisovanie vysledkov
52            if (t >= saveStep)
53                  {
54                    // zapisanie dat
55                    vtkout(grid,c,"concentration",counter);
56
57                    // zvysit pocitadlo a saveStep pre dalsi interval
58                    saveStep += saveInterval;
59                    ++counter;
60                  }
61
62            // info o mriezke case, casovom kroku
63            std::cout << "s=" << grid.size(0) <<
64            " k=" << k << " t=" << t << " dt=" << dt << std::endl;
65          }
66    // chyba po poslednom casovom kroku
67    chyba(grid,mapper,c,cp);
68
69    // zapis vysledkov
70    vtkout(grid,c,"concentration",counter);
71  }
72
73  // hlavny program
74  int main (int argc , char ** argv)
75  {
76    int I = 0, z = 0;
77    std::cout <<"Pocet elementov: "<< std::endl;
78    std::cin >>I;
79    std::cout <<"Stupen zjemnenia: "<< std::endl;
80    std::cin >>z;
81
82    // DUNE vypise hlasku ak nastane vynimka
83    try {
84      using namespace Dune;
85
86      // vytvorenie siete
87       const int dim=2;
88       typedef Dune::SGrid<dim,dim> GridType;
89       Dune::FieldVector<int,dim> N(I);
90       Dune::FieldVector<GridType::ctype,dim> L(0.0);
91       Dune::FieldVector<GridType::ctype,dim> H(1.0);
92       GridType grid(N,L,H);
93       grid.globalRefine(z);
94
95      // vypocet do casu 8.0
96      timeloop(grid, 8.0);
97    }
98    catch (std::exception & e) {
99      std::cout << "STL ERROR: " << e.what() << std::endl;
100     return 1;
101   }
102   catch (Dune::Exception & e) {
103     std::cout << "DUNE ERROR: " << e.what() << std::endl;
104     return 1;
105   }
106   catch (...) {
107     std::cout << "Unknown ERROR" << std::endl;
108     return 1;
109   }
110
111   return 0;
112 }
```

Funkcia `timeloop` vytvorí zobrazovač a alokuje vektor riešení, v ktorom jeden prvok je jedna hodnota riešenia na sieti (hodnota vypočítaná na jednom elemente). Tento vektor sa inicializuje na riadku 27 kde sa vypočítajú riešenia z počiatočnej podmienky. Funkcia `vtkout` zapisuje vypočítané dáta do súboru vo formáte VTK, ktorý je možné použiť napr. v softvéri Paraview na vizualizáciu výsledkov. V cykle `while` je volaná funkcia `evolve`, ktorá realizuje samotný výpočet. Po poslednom časovom kroku je vypočítaná L1 chyba.

V hlavnom programe zadávame počet elementov časový krok a stupeň zjemnenia siete pričom jedno zjemnenie znamená rozdelenie každého elementu na polovicu v každom smere.

5.2 Metóda druhého rádu počítaná na elementoch

Pri použití metódy druhého rádu upravíme vo funkcii `evolve` riadky 80-122 čo je implementácia rovnice (19):

```
1  if (is.neighbor())
2   {
3    // pristup na suseda
4    EntityPointer outside = is.outside();
5    int indexj = mapper.map(*outside);
6
7    // vypocet tokov
8    if ( it->level()>outside->level() ||
9     (it->level()==outside->level() && indexi<indexj) )
10     {
11       // vypocet sucinu normaloveho vektora a rychlosti
12       Dune::GeometryType nbgt = outside->type();
13       const Dune::FieldVector<ct,dim>&
14         nblocal = Dune::ReferenceElements<ct,dim>::general(nbgt).position(0,0);
15       double nbvolume = outside->geometry().integrationElement(nblocal)
16         *Dune::ReferenceElements<ct,dim>::general(nbgt).volume();
17       double nbfactor = velocity*integrationOuterNormal/nbvolume;
18
19       // globalne suradnice stredu suseda
20       Dune::FieldVector<ct,dimworld>
21          nbglobal = it->geometry().global(nblocal);
22
23       // rychlost v strede suseda
24          Dune::FieldVector<double,dim> velocitynbglobal = u(nbglobal,t+0.5*dt);
25
26       if (factor<0) // tok dovnutra
27         {
28           dd = gr[indexj].h[0]*(faceglobal[0]-nbglobal[0]) +
29             gr[indexj].h[1]*(faceglobal[1]-nbglobal[1]) - (dt/2.0) *
30             (gr[indexj].h[0]*velocitynbglobal[0] +
31             gr[indexj].h[1]*velocitynbglobal[1]);
32
33           update[indexi] -= ( c[indexj] + dd ) * factor;
34           update[indexj] += ( c[indexj] + dd ) * nbfactor;
35         }
36          else // tok von
37         {
38           db = gr[indexi].h[0]*(faceglobal[0]-global[0]) +
39             gr[indexi].h[1]*(faceglobal[1]-global[1]) - (dt/2.0) *
40             (gr[indexi].h[0]*velocityglobal[0] +
41             gr[indexi].h[1]*velocityglobal[1]);
42
```

```
43            update[indexi] -= ( c[indexi] + db ) * factor;
44            update[indexj] += ( c[indexi] + db ) * nbfactor;
45           }
46       }
47   }
48
49 // ak hranica oblasti
50 if (is.boundary())
51   {
52     if (factor<0) // aplikacia okrajovej podmienky
53     update[indexi] -= b(faceglobal,t+0.5*dt)*factor;
54
55     else // outflow
56      {
57        db = gr[indexi].h[0]*(faceglobal[0]-global[0]) +
58          gr[indexi].h[1]*(faceglobal[1]-global[1]) - (dt/2.0) *
59          (gr[indexi].h[0]*velocityglobal[0] +
60          gr[indexi].h[1]*velocityglobal[1]);
61
62     update[indexi] -= (c[indexi] + db)* factor;
63
64      }
65 }
```

5.3 Metóda prvého rádu počítaná na vrcholoch

Vypočítané hodnoty teraz ukladáme pre vrcholy siete. Cyklus cez hrany už nie je realizovaný pomocou IntersectionIterator. Využili sme referenčný element a jeho podentity vyššej kodimenzie (riadky 63-137). Keďže tok na hranici výpočtovej oblasti, v prípade príkladu single vortex, je nulový, jeho výpočet neuvádzame. Hodnoty gradientu budú opäť ukladané do vektora `gr`.

```
1  #include<dune/grid/common/referenceelements.hh>
2
3  template<class G, class M, class V, class Z>
4  void evolve (const G& grid, const M& mapper, V& c, double t, double& dt, Z& gr)
5  {
6    const int dim = G::dimension;
7    const int dimworld = G::dimensionworld;
8
9    typedef typename G::ctype ct;
10
11   const typedef typename G::template Codim<0>::LeafIndexSet& LeafIndexSet;
12
13   // vytvorime si potrebne  typy ukazovatelov s konkretnymi
14   // nazvami, pre jednoduchsiu orientaciu
15   typedef typename G::template Codim<0>::LeafIterator ElementIterator;
16
17   typedef typename G::template Codim<dim-1>::EntityPointer EdgePointer;
18
19   typedef typename G::template Codim<dim>::EntityPointer VertexPointer;
20
21   // na ziskanie indexov vrcholov hrany nepouzijeme mapper, ale
22   // priamo mnozinu indexov Dune::IndexSet
23   LeafIndexSet indexset = grid.leafIndexSet();
24
25   V update(c.size());
26   for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;
27
28   // na plochu okolo vrcholu, ktorou treba vypocitanu hodnotu predelit
29   std::vector<double> area(c.size());
30           for (unsigned int i = 0; i < area.size(); i++) area[i] = 0;
31
```

```
32    // cyklus cez elementy
33    ElementIterator endit = gridView.template end<0>();
34    for (ElementIterator it = gridView.template begin<0>(); it!=endit; ++it)
35          {
36            // geometria elementu
37            Dune::GeometryType gt = it->type();
38
39            // stred elementu v lokalnych suradniciach
40            const Dune::FieldVector<ct,dim>&
41                  local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
42
43            // stred elementu v globalnych suradniciach
44            Dune::FieldVector<ct,dimworld>
45                  elementCenter = it->geometry().global(local);
46
47            // velkost elementu
48            double volumeElement = it->geometry().integrationElement(local)
49                  *Dune::ReferenceElements<ct,dim>::general(gt).volume();
50
51            // pocet vrcholov
52            int numberVertices = it->geometry().corners();
53
54            // plocha okolo vrcholu
55            for (int i = 0; i < numberVertices; i++)
56                  {
57                    VertexPointer vp = it->template entity<dim>(i);
58
59                    area[mapper.map(*vp)] += volumeElement / numberVertices;
60                  }
61
62            // cyklus cez hrany
63            int eend = Dune::ReferenceElements<ct,dim>::general(gt).size(dim-1);
64            for (int edge=0; edge<eend; edge++)
65                  {
66                    // lokalne indexy vrcholov hrany
67                    int startVertex =
68                     Dune::ReferenceElements<ct,dim>::general(gt).subEntity(edge,dim-1,0,dim);
69                    int endVertex =
70                     Dune::ReferenceElements<ct,dim>::general(gt).subEntity(edge,dim-1,1,dim);
71
72                    // globalne indexy vrcholov hrany
73                    int indexi = indexset.template subIndex<dim>(*it,startVertex);
74                    int indexj = indexset.template subIndex<dim>(*it,endVertex);
75
76                    // lokalna suradnica startVertex-u
77                    const Dune::FieldVector<ct,dim>&
78                          svLocal =
79                      Dune::ReferenceElements<ct,dim>::general(gt).position(startVertex,dim);
80
81                    // globalna suradnica startVertex-u
82                    Dune::FieldVector<ct,dimworld>
83                          svGlobal = it->geometry().global(svLocal);
84
85                    // lokalna suradnica endVertex-u
86                    const Dune::FieldVector<ct,dim>&
87                          evLocal =
88                      Dune::ReferenceElements<ct,dim>::general(gt).position(endVertex,dim);
89
90                    // globalna suradnica endVertex-u
91                    Dune::FieldVector<ct,dimworld>
92                          evGlobal = it->geometry().global(evLocal);
93
94                    // ukazovatel na hranu
95                    EdgePointer ep = it->template entity<dim-1>(edge);
96
97                    // stred hrany v lokalnych suradniciach
98                    const Dune::FieldVector<ct,dim-1>&
99                          edgeCenterLocal =
100                     Dune::ReferenceElements<ct,dim-1>::general(ep->type()).position(0,0);
```

```
101
102                     // stred hrany v globalnych suradniciach
103                     Dune::FieldVector<ct,dimworld> edgeCenter =
104                           ep->geometry().global(edgeCenterLocal);
105
106                     // stred segmentu
107                     Dune::FieldVector<ct,dim> segmentCenter = edgeCenter + elementCenter;
108                     segmentCenter[0] /= 2.0;
109                     segmentCenter[1] /= 2.0;
110
111                     // rychlost na segmente
112                     Dune::FieldVector<ct,dim> velocity = u(segmentCenter,t);
113
114                     // pomocny vektor
115                     Dune::FieldVector<ct,dim> help = elementCenter - edgeCenter;
116
117                     // normalovy vektor na segmente
118                     Dune::FieldVector<ct,dim> normal;
119                     normal[0] = fabs(help[1]);
120                     normal[1] = fabs(help[0]);
121
122                     // tok
123                     double flux = (normal[0] * velocity[0]) + (normal[1] * velocity[1]);
124
125                     if (flux<0)
126                       {
127                           update[indexi] -= flux * c[indexj];
128                           update[indexj] += flux * c[indexj];
129                       }
130
131                     else
132                       {
133                           update[indexi] -= flux * c[indexi];
134                           update[indexj] += flux * c[indexi];
135                       }
136
137                   } // koniec cez hrany
138
139           } // koniec cez mriezku
140
141     for (unsigned int i=0; i<c.size(); ++i)
142           c[i] += dt*update[i] / area[i];
143
144     return;
145   }
```

5.4 Metóda druhého rádu počítaná na vrcholoch

Gradient je teraz implementovaný vo funkcii `grad_evolve` takto:

```
1   template<class G, class M, class V, class C>
2   void grad_eval(const G& grid, const M& mapper, V& c, C& gr)
3   {
4     const int dim = G::dimension;
5
6     // typ suradnic
7     typedef typename G::ctype ct;
8
9     // pohlad
10    typedef typename G::LeafGridView GridView;
11
12    // typ iteratora cez elementy
13    typedef typename GridView::template Codim<0>::Iterator LeafIterator;
14
15    // typ iteratora cez vrcholy
```

```
16    typedef typename GridView::template Codim<dim>::Iterator VertexIterator;
17
18    // typ ukazovatela na vrchol
19    typedef typename G::template Codim<dim>::EntityPointer VertexPointer;
20
21    typedef Dune::FieldVector<ct,dim> DuneVector;
22
23    // na plochu okolo vrcholu
24    std::vector<double> area(c.size());
25          for (unsigned int i = 0; i < area.size(); i++) area[i] = 0;
26
27    for (int i = 0; i < gr.size(); i++)
28          {
29            gr[i][0] = 0; gr[i][1] = 0;
30          }
31
32    // cyklus cez celu siet
33    LeafIterator endit = gridView.template end<0>();
34    for (LeafIterator it = gridView.template begin<0>(); it!=endit; ++it)
35          {
36            // typ geometrie elementu
37            Dune::GeometryType gt = it->type();
38
39            // pocet vrcholov
40            int numberVertices =
41                  Dune::ReferenceElements<ct,dim>::general(gt).size(dim);
42
43            // stred elementu v lokalnych suradniciach
44            const Dune::FieldVector<ct,dim>&
45                  local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
46
47            // velkost elementu
48            double volumeElement = it->geometry().integrationElement(local)
49                  *Dune::ReferenceElements<ct,dim>::general(gt).volume();
50
51            // ak stvoruholnik
52            if (gt.isQuadrilateral())
53              {
54                  // na globalne hodnoty
55                  Dune::FieldVector<ct,numberVertices> value;
56                     for (int i = 0; i < numberVertices; i++) value[i] = 0.0;
57
58                  // na lokalny gradient
59                  Dune::FieldVector<DuneVector,numberVertices> gradientLocal;
60
61                  for (int i = 0; i < numberVertices; i++)
62                          {
63                            gradientLocal[i][0] = 0.0;
64                            gradientLocal[i][1] = 0.0;
65
66                            // ziskanie hodnot na vrcholoch
67                            VertexPointer vp = it->template entity<dim>(i);
68                            value[i] = c[mapper.map(*vp)];
69                          }
70
71                  // lokalny gradient
72                  gradientLocal[0][0] = value[1] - value[0];
73                  gradientLocal[0][1] = value[2] - value[0];
74
75                  gradientLocal[1][0] = value[1] - value[0];
76                  gradientLocal[1][1] = value[3] - value[1];
77
78                  gradientLocal[2][0] = value[3] - value[2];
79                  gradientLocal[2][1] = value[2] - value[0];
80
81                  gradientLocal[3][0] = value[3] - value[2];
82                  gradientLocal[3][1] = value[3] - value[1];
83
84
```

```
85                  for (int vertex = 0; vertex < numberVertices; vertex++)
86                    {
87                      VertexPointer vp = it->template entity<dim>(vertex);
88                      int index = mapper.map(*vp);
89
90                      // lokalne suradnice vrchola
91                      const Dune::FieldVector<ct,dim>& vertexLocal =
92                       Dune::ReferenceElements<ct,dim>::general(gt).position(vertex,dim);
93
94                      // jacobian globalneho uzla
95                      const Dune::FieldMatrix<ct,dim,dim>& jacobi =
96                          it->geometry().jacobianInverseTransposed(vertexLocal);
97
98                      // globalny gradient = lokalny gradient vynasobeny inverznym
99                      // jakobianom a plochou prisluchajucou k vrcholu
100                     gr[index][0] += (volumeElement / numberVertices) *
101                         ((jacobi[0][0] * gradientLocal[vertex][0]) +
102                         (jacobi[0][1] * gradientLocal[vertex][1]));
103
104                     gr[index][1] += (volumeElement / numberVertices) *
105                         ((jacobi[1][0] * gradientLocal[vertex][0]) +
106                         (jacobi[1][1] * gradientLocal[vertex][1]));
107
108                   } // koniec for
109
110             } // koniec if gt.isQuadrilateral()
111
112           // plocha okolo vrcholu
113           for (int vertex = 0; vertex < 4; vertex++)
114                 {
115                   VertexPointer vp = it->template entity<dim>(vertex);
116                   area[mapper.map(*vp)] += volumeElement / numberVertices;
117                 }
118
119         } // koniec cez vsetky elementy
120
121         // predelime globalny gradient plochou okolo vrcholu
122         for (VertexIterator vi = grid.template leafbegin<dim>();
123           vi!=grid.template leafend<dim>(); ++vi)
124                 {
125                   gr[mapper.map(*vi)][0] /= area[mapper.map(*vi)];
126                   gr[mapper.map(*vi)][1] /= area[mapper.map(*vi)];
127                 }
128 }
```

Vo funkcii `evolve` sa oproti metóde prvého rádu zmenia riadky 123-135 takto:

```
1  if (flux<0)
2    {
3      dd = gr[indexj][0] * (segmentCenter[0] - evGlobal[0]) +
4           gr[indexj][1] * (segmentCenter[1] - evGlobal[1]) -
5           0.5 * dt * ( u(evGlobal,t+0.5*dt)[0] *
6           gr[indexj][0] + u(evGlobal,t+0.5*dt)[1] * gr[indexj][1] );
7
8      update[indexi] -= flux * (c[indexj] + dd);
9      update[indexj] += flux * (c[indexj] + dd);
10   }
11
12 else
13   {
14     db = gr[indexi][0] * (segmentCenter[0] - svGlobal[0]) +
15          gr[indexi][1] * (segmentCenter[1] - svGlobal[1]) -
16          0.5 * dt * ( u(svGlobal,t+0.5*dt)[0] *
17          gr[indexi][0] + u(svGlobal,t+0.5*dt)[1] * gr[indexi][1] );
18
19     update[indexi] -= flux * (c[indexi] + db);
20     update[indexj] += flux * (c[indexi] + db);
21   }
```

6 Zhodnotenie dosiahnutých výsledkov a navrhnutých riešení

V nasledujúcich tabuľkách uvádzame vypočítané L1 chyby po použití oboch metód na elementoch pri N časových krokoch a deleniach siete 50, 100 a 150. Obdobné výsledky sme získali aj po použití metód na vrcholoch.

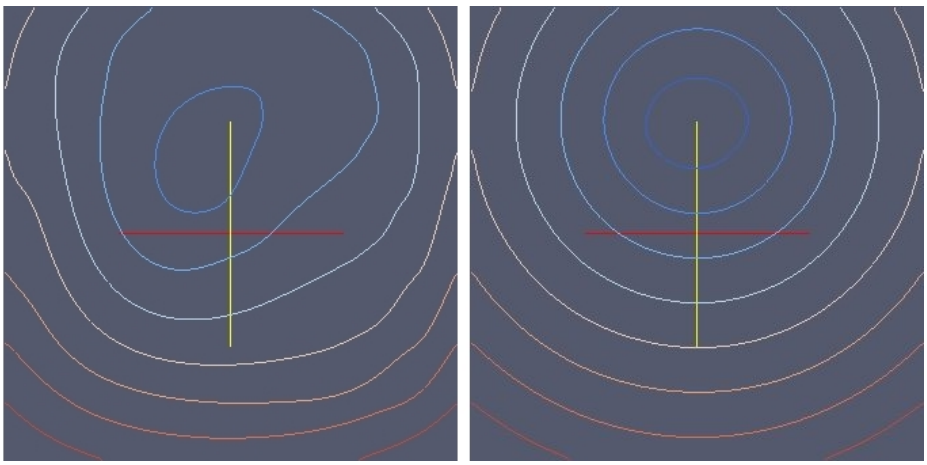
I	N	chyba
50	500	0.0555
100	1000	0.0370
150	1500	0.0280

Tabuľka 1: Chyby po použití metódy prvého rádu na elementoch.

I	N	chyba
50	500	0.00604
100	1000	0.00186
150	1500	0.00080

Tabuľka 2: Chyby po použití metódy druhého rádu na elementoch.

Vidíme, že chyba pri použití metódy prvého rádu konverguje s rádom 1 a metóda druhého rádu, ktorú sme odvodili, konverguje s rádom 2. Pre vizuálne porovnanie ešte uvádzame (Obrázku 2.) vykreslené riešenia v časoch $t = 8$ pri delení siete $I = 150$ po použití oboch metód.



Obrázok 8: Kontúry numerického riešenia príkladu single vortex v časoch $t = 8$.

Súhrn

Cieľom práce bol návrh a realizácia numerickej metódy druhého rádu presnosti na riešenie rovnice advekcie pre dané rýchlostné vektorové pole. Dôraz bol kladený na počítačovú realizáciu v modernom softvéri **DUNE** (Distributed and Unified Numerics Environment). Metódu sme úspešne implementovali, podrobne popísali a otestovali na netriviálnom príklade. Implementácia bola zrealizovaná aj pre metódu konečných objemov s neznámymi vo vrcholoch siete, ktorá doteraz v softvéri **DUNE** nebola implementovaná. Práca poskytuje úvodné informácie potrebné pre implementáciu podobných metód konečných objemov na riešenie parciálnych diferenciálnych rovníc v softvéri **DUNE**.

Summary

The aim of this work was a proposal and realization of a second order accurate numerical method for the solution of advection equation for a given velocity vector field. The emphasize has been given on the implemenntation of such method in modern software tool **DUNE** (Distributed and Unified Numerics Environment). The method was successfully realized, described in details and tested on a nontrivial example. The implementation was done also for a finite volume method with unknowns placed in vertices of a grid that was not available in software **DUNE**. This bachelor work offers introductory information for analogous implementation of similar finite volume methods for solution of partial differential equations in software **DUNE**.

Referencie

[1] Peter Frolkovič, Christian Wehner: *Flux-based level set method on rectangular grids and computation of first arrival time functions*, Computing and Visualization in Science, DOI 10.1007/s00791-008-0115-z, 2008

[2] Peter Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, Robert Klöfkorn, Martin Nolte, Mario Ohlberger, Oliver Sander: *The Distributed and Unified Numerics Environment (DUNE) Grid Interface HOWTO*, Version 1.3svn, March 10, 2009

[3] [Bastian et al. (2008)] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander: A Generic Grid Interface for Parallel and Adaptive Scientific Computing. *Part I: Abstract Framework. Submitted to Computing*

[4] [Bastian et al. (2008)] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander: A Generic Grid Interface for Parallel and Adaptive Scientific Computing. *Part II: Implementation and Tests in DUNE. Submitted to Computing.*

[5] **DUNE** class documantation (`http://www.dune-project.org/doc/doxygen.html`)

Prílohy

Na priloženom CD sa nachádza aktuálna verzia modulu `dune-common`, modulu `dune-grid` a upraveného modulu `dune-grid-howto` s názvom `M2_dualna_siet.tar.gz`, ktorý obsahuje ukážkové programy a implementácie metódy druhého rádu počítanej na vrcholoch. Na CD sa nachádzajú aj dve animácie príkladu single vortex. Pri prvej animácii `M1_150.avi` bol príklad vypočítaný metódou prvého rádu na elementoch a pri druhej animácii `M2_150.avi` bol príklad počítaný metódou druhého rádu na elementoch.