SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Stavebná fakulta

Implementácia semi-implicitných metód konečných objemov pre spracovanie obrazu pomocou knižníc ITK

Diplomová práca

SvF-5343-39169

Študíjny program:	matematicko-počítačové modelovanie
Pracovisko (katedra/ústav):	KMDG
Vedúci záverečnej práce:	Mgr. Mariana Remešíková, PhD.

Bratislava 2011 **Bc. Juraj Schiffer**

Čestné prehlásenie

Vyhlasujem, že som diplomovú prácu vypracoval samostatne s použitím uvedenej odbornej literatúry.

Bratislava 20. mája 2011 .....

Vlastnoručný podpis

Poďakovanie

Poďakovanie vedúcej diplomovej práce za odbornú pomoc a pripomienky pri jej vypracovávaní, rodine a priateľom za podporu a pochopenie.

Súhrn

Ciele tejto diplomovej práce je možné rozdeliť na tri časti. Prvou úlohou je naštudovať semi-implicitné metódy numerického riešenia parciálnych diferenciálnych rovníc (PDR), vysvetliť a názorne ukázať ich využitie pri spracovaní obrazu. Ďalej je potrebné popísať a porovnať ich rozdiely oproti iným používaným metódam (explicitné, implicitné). Druhou úlohou je zoznámenie sa so systémom knižníc ITK, pochopiť a následne popísať jednotlivé princípy tohto rozsiahleho systému. Treťou a najdôležitejšou časťou práce je implementácia semi-implicitných metód na spracovanie obrazu spôsobom, ktorý by umožnil ich prípadné zaradenie do systému knižníc ITK. Pritom je zámerom čo najefektívnejšie využiť prostriedky, ktoré nám ITK ponúka.

Kľúčové slová: semi-implicitné metódy, spracovanie obrazu, Insgiht Toolkit (ITK)

Abstract

The goals of this diploma thesis can be divided in three parts. The first task is to study semi-implicit methods for numerical solution of PDE's, to explain and illustrate their use in image processing. The next thing to do is to describe and compare differences with other known methods (explicit, implicit). The second task is to get familiar with the system of ITK libraries, understand and describe the main principles of this extensive system. The last but not the least part of the thesis is the implementation the semi-implicit methods for image processing in a way that would enable eventual incorporation into the system of ITK libraries. The goal is to use the components, which ITK offers to us in the most efficient way possible.

Key words: semi-implicit methods, image processing, Insight Toolkit (ITK)

Obsah

Úvod

V tomto úvode by som rád podrobnejšie rozobral názov témy tejto práce: *Implementácia semi-implicitných metód konečných objemov pre spracovanie obrazu pomocou knižníc ITK*.

A prečo nezačať od konca. V prvej kapitole sa trochu rozpíšeme o *knižniciach ITK*. Z tejto kapitoly sa čitateľ dozvie o hlavných myšlienkach tohto rozsiahleho softvérového balíka. So samotnými myšlienkami toho veľa neurobíme, preto sa vyjadríme k spôsobu, ktorým sa tieto myšlienky pretvárajú na skutočnosť do samotného kódu softvéru.

Druhá časť názvu, *semi-implicitné metódy konečných objemov pre spracovanie obrazu* je rozpracovaná v druhej kapitole. Základným kameňom tejto kapitoly je matematický model, v našom prípade bude reprezentovaný rovnicou. Takýchto modelov bude v práci rozpísaných viac. Spoločnou vlastnosťou týchto modelov je ich uplatnenie v *spracovaní obrazu*. Postupne popíšeme krok za krokom, ktoré nás dovedú od nelineárnej parciálnej diferenciálnej rovnice až k vytúženému výsledku, diskrétnej schéme na výpočet numerického riešenia. Medzi tieto kroky bude patriť *semi-implicitná* časová diskretizácia a použitie *metódy konečných objemov* pri priestorovej diskretizácii.

Tretia kapitola, *implementácia* spája myšlienky obsiahnuté v predchádzajúcich dvoch kapitolách. Popisuje spôsob, ktorým sme implementovali matematické modely spracovania obrazu s využitím tried knižníc ITK.

Kapitola 1

ITK

Zdrojom informácií pre túto kapitolu bola dokumentácia ITK [1].

ITK (The Insight Toolkit) je softvérový balík na spracovanie obrazu (filtrácia, segmentácia, štatistika a pod.) patriaci do skupiny projektov s otvoreným kódom (open-source). ITK predstavuje súbor knižníc implementovaných v C++. Na bezproblémovú kompiláciu ITK na rôznych platformách slúži program CMake. Taktiež pomocou programu Cable je možné využívať knižnice ITK v iných programovacích jazykoch. Vývoj ITK sa nesie v duchu generického programovania, pri ktorom sa používajú šablóny, aby ten istý kód mohol byť využitý pre rôzne dátové typy. Výhodou generického programovania je vysoko univerzálny kód. Pri vývoji ITK sa využíva model, pri ktorom je tvorba softvéru neustály iteratívny proces návrhu, implementácie, testovania a vydania. Dôležitú úlohu pri tomto spôsobe vývoja softvéru zohráva komunikácia a testovanie. Nie menej dôležitou časťou každého softvéru je dokumentácia. ITK využíva na vytváranie dokumentácie systém Doxygen, ktorého výstupy sú prístupné pre širokú verejnosť.

1.1 Základné princípy ITK

1.1.1 Generické programovanie

Generické programovanie je metóda pri ktorej program obsahuje generické komponenty. Myšlienkou generického programovania je schopnosť vytvárať efektívny a adaptívny kód. Generické programovanie je implementované v C++ pomocou šablón a STL (Standard Template Library).

Šablóny umožňujú používateľovi písať program pre viac neznámych dátových typov T naraz. Pred samotnou kompiláciou kódu, používateľ softvéru musí špecifikovať dátový typ T. Dátový typ T môže byť preddefinovaný alebo

používateľom definovaný. Počas kompilácie, kompilátor vyhodnocuje zhodnosť dátového typu T so šablónou a jeho podporu použitými metódami a operátormi.

ITK využíva generické programovanie priamo vo svojom jadre. Ako príklad sa dá uviesť vytvorenie obrazu s rôznym typom pixelu (unsigned char, float, RGB).

Triedy ITK definujú 2 súbory: hlavičkový .h a implementačný súbor .cxx pre prípad triedy bez použitia šablón a .txx ak trieda využíva šablóny.

1.1.2 Reprezentácia dát

Dátové typy ITK uchovávajúce údaje delíme na dve veľké skupiny: obraz a sieť. Obidva dátové typy sú podtriedou triedy itk::DataObject.

Trieda itk::Image predstavuje N-rozmerné pole dát. Väčšinou je používaná na reprezentáciu obrazov, ale vo všeobecnosti môže ísť o ľubovoľné pole údajov. Pri vytváraní inštancie je potrebné špecifikovať dátový typ TPixel, ktorý predstavuje typ 1 prvku poľa a dimenzionalitu obrazu. Podrobnejšie sa triedou itk::Image budeme zaoberať v kapitole 1.2.

Trieda itk::Mesh reprezentuje N-rozmernú neštruktúrovanú sieť.

1.1.3 Proces spájania objektov

Kým dátové objekty (obraz, sieť) sú používané na reprezentáciu údajov, výkonné objekty sú triedy, ktoré vykonávajú operácie na dátových objektoch a taktiež môžu vytvárať nové dátové objekty.

Najbežnejší spôsob prepojenia dátových a výkonných objektov je použitie metód SetInput(), GetOutput(), Update(). Pri ich zavolaní vykonajú preddefinovaný sled operácií, ktoré zahŕňajú tak dátové ako aj výkonné objekty. Výhodou týchto metód je, že predtým ako sa spustia, overia, či nastala zmena oproti ich poslednému spusteniu a až po tomto úkone vykonajú všetky definované operácie.

1.2 Obraz

Trieda itk::Image, ako bolo spomenuté vyššie, predstavuje triedu na uchovávanie údajov. Z názvu triedy vyplýva, že najčastejšie budú týmito údajmi obrazové dáta.

Na vytvorenie a prácu s obrazom potrebujeme ako prvé zahrnúť do programu hlavičkový súbor pre triedu itk::Image.

```
#include "itkImage.h"
```

Trieda itk::Image je generická trieda, parametrami šablóny sú dátový typ obrazového bodu a jeho dimenzionalita. Na zjednodušenie ďalšej práce si zadefinujeme obrazový typ ImageType, kde v zobrazenom príklade bude dimenzionalita rovná 3 a dátový typ unsigned short.

```
typedef itk::Image< unsigned short, 3 > ImageType;
```

Obraz sa vytvára zavolaním metódy New() zodpovedajúceho obrazového typu ImageType. Funkcia vráti smerník na daný obraz, ktorý sa zapíše do novej premennej image. Táto premenná bude predstavovať samotný obraz.

```
ImageType::Pointer image = ImageType::New();
```

Veľkosť obrazu je definovaná pomocou dvoch parametrov typu: IndexType a SizeType. IndexType predstavuje umiestnenie počiatku v obraze. Zvyčajne to je nulový vektor. SizeType definuje veľkosť oblasti. IndexType je reprezentovaný N-rozmerným poľom, kde každý prvok predstavuje súradnicu počiatočného pixelu v prislúchajúcom smere.

```
ImageType::IndexType start;
start[0] = 0;
start[1] = 0;
start[2] = 0;
```

SizeType je taktiež reprezentovaný N-rozmerným poľom, kde prvky poľa určujú veľkosť obrazu v smere každej osi.

```
ImageType::SizeType size;
size[0] = 200;
size[1] = 200;
size[2] = 200;
```

Ďalej nasleduje samotná inicializácia oblasti, ktorú predstavuje typ RegionType. V tomto prípade je oblasť celým výsledným obrazom, avšak vďaka oddeleniu obrazu a oblasti je možné zvoliť si za oblasť len časť obrazu, s ktorou pracujeme a tým znížiť napríklad pamäťové nároky.

```
ImageType::RegionType region;
region.SetSize( size );
region.SetIndex( start );
```

Nakoniec je vyššie definovaná oblasť použitá na definovanie samotného obrazu. Treba si uvedomiť, že zatiaľ nebola alokovaná žiadna pamäť pre obrazové dáta. Na túto úlohu slúži metóda Allocate().

```
image->SetRegions( region );
image->Allocate();
```

V praxi sa obrazy často nevytvárajú manuálne, ale sú získavané z externého zdroja (súbor). Na načítanie obrazu zo súboru je potrebné vložiť hlavičkový súbor triedy itk::ImageFileReader.

```
#include "itkImageFileReader.h"
```

Znovu si definujeme typ obrazu (typ pixelu a dimenzionalitu).

```
typedef itk::Image< unsigned short, 3 > ImageType;
```

Následne si zadefinujeme typ „čítača“, pričom ako parameter do šablóny vstupuje typ obrazu.

```
typedef itk::ImageFileReader< ImageType > ReaderType;
```

Zavolaním metódy New() zodpovedajúceho typu „čítača“ ReaderType získame smerník daného typu, ktorý zapíšeme do premennej reader, ktorý predstavuje nášho „čítača“.

```
ReaderType::Pointer reader = ReaderType::New();
```

Minimálna vyžadovaná informácia potrebná na načítavanie zo súboru je meno súboru. Táto informácia sa nastavuje pomocou metódy SetFileName().

```
reader->SetFileName( "filename" );
```

Samotné načítanie vyvoláme pomocou metódy Update().

```
reader->Update();
```

Prístup k načítanému obrazu získame zavolaním metódy GetOutput().

```
ImageType::Pointer image = reader->GetOutput();
```

1.3 Obrazový iterátor

V tejto kapitole opíšem obrazový iterátor, dôležitú súčasť generického programovania pre spracovanie obrazov v ITK. Iterátor je zovšeobecnenie známeho smerníka z programovacieho jazyka C, používaného na uchovávanie adries údajov v pamäti.

Model generického programovania definuje funkčne nezávislé komponenty: kontajnery (dátové štruktúry) a algoritmy. Kontajnery uchovávajú dáta a algoritmy vykonávajú operácie na dátach. Algoritmy využívajú na prístup k dátam v kontajneroch tretiu triedu, iterátory.

V ITK sú obrazové iterátory špeciálne navrhnuté na prácu s obrazovými kontajnermi s rôznym typom pixelu, príp. dimenzie, preto ich rozhranie a implementácia je optimalizovaná pre úlohy spracovania obrazu. Použitie obrazových iterátorov namiesto pristupovania k dátam priamo má mnoho výhod. Kód je kompaktnejší a zovšeobecňuje použitie na rôzne dimenzie, algoritmy bežia rýchlejšie a iterátory zjednodušujú úlohy, ako multithreading a metódy spracovania obrazu, v ktorých sa využívajú prvky okolia daného obrazového bodu.

Najbežnejší obrazový iterátor ITK je reprezentovaný triedou itk::ImageRegionIterator. Tento typ iterátora patrí medzi najmenej špecializované, preto je optimalizovaný na rýchlosť iterovania a výhodný pre operácie, pri ktorých nie je dôležitá informácia o pozícii iterátora v obraze.

Existujú dve verzie obrazových iterátorov: non-const a const. Najväčším rozdielom je, že const verzia dokáže len čítať dáta a nemôže zapisovať dáta.

Nasleduje jednoduchý príklad na definíciu základného obrazového iterátora pre itk::Image. Podobne ako v prípade definície ImageType v predchádzajúcej kapitole, zadefinujeme si ConstIteratorType a IteratorType zo zodpovedajúcich tried itk::ImageRegionConstIterator a itk::ImageRegionIterator.

Každý obrazový iterátor má najmenej jeden parameter šablóny, je ním typ obrazu cez ktorý iteruje. V tomto prípade nie je žiadne obmedzenie na typ pixelu alebo dimenziu obrazu.

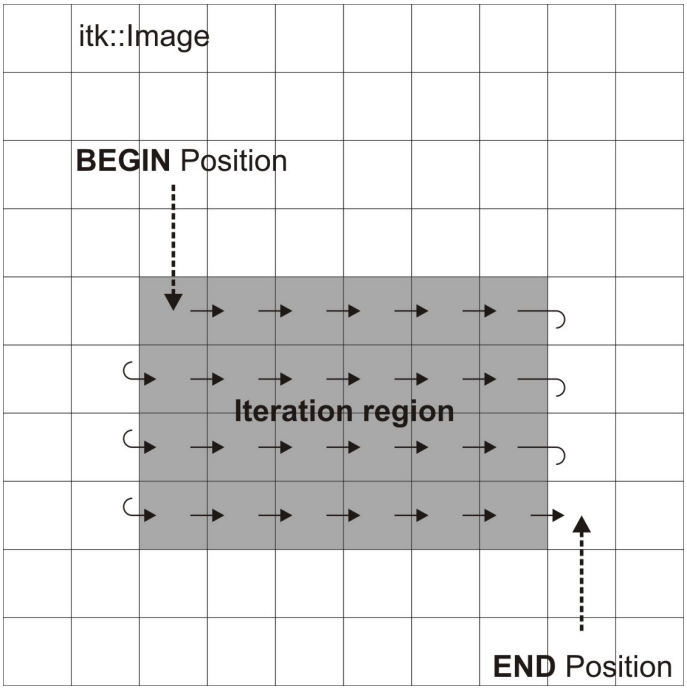
```
typedef itk::Image<float, 3> ImageType;
typedef itk::ImageRegionConstIterator< ImageType >
    ConstIteratorType;
typedef itk::ImageRegionIterator< ImageType > IteratorType;
```

Ďalej si načítame obraz do premennej image

```
ImageType::Pointer image = SomeFilter->GetOutput();
```

kde za „SomeFilter“ môžeme dosadiť napr. „reader“ z predchádzajúcej kapitoly.

Konštruktor iterátora vyžaduje aspoň dva argumenty: smerník na obraz a oblasť obrazu, cez ktorú sa iteruje. Táto oblasť obrazu sa nazýva iteračná oblasť a musí byť celá obsiahnutá v obraze. Inak povedané, iteračná oblasť je podoblasť obrazu. V našich ďalších prípadoch bude iteračná oblasť totožná s obrazom, preto môžeme využiť metódu GetRequestedRegion() triedy itk::Image.



Obr. 1.1: Zobrazenie iteračnej oblasti, začiatočnej a koncovej pozície iterátora

```
ConstIteratorType
    constIterator( image, image->GetRequestedRegion() );
IteratorType iterator( image, image->GetRequestedRegion() );
```

Pohyb iterátora cez iteračnú oblasť môže prebiehať dvoma spôsobmi. Dopredná iterácia (Obr.1.1) ide zo začiatku iteračnej oblasti na jej koniec. Spätná iterácia ide z pozície za koncom oblasti späť na jej začiatok. Iterátor sa môže presunúť na jednu z týchto štartovacích pozícii pomocou nasledujúcich dvoch metód.

- GoToBegin() - nasmeruje iterátor na prvý platný element oblasti
- GoToEnd() - nasmeruje iterátor jednu pozíciu za posledným platným elementom oblasti

Pohyb iterátorov cez oblasť zabezpečuje operátor zvyšovania a znižovania.

- operátor++() - zvyšuje pozíciu iterátora o jedna v kladnom smere.
- operátor−−() - znižuje pozíciu iterátora o jedna v zápornom smere.

Na zastavenie pohybu cez iteračnú oblasť slúžia metódy IsAtEnd() a IsAtBegin() vracajúce hodnotu typu bool. Obrazový iterátor taktiež dokáže odovzdať svoju aktuálnu pozíciu v obraze pomocou metódy GetIndex().

Obrazové iterátory definujú dve základné metódy na čítanie a zápis hodnôt pixelu.

- Get() s návratovou hodnotou typu PixelType, vracia hodnotu pixelu na aktuálnej pozícii iterátora.
- Set(PixelType) nastaví hodnotu pixel na aktuálnej pozícii iterátora na hodnotu argumentu typu PixelType. Táto metóda nie je definovaná pre konštantné verzie iterátorov.

Metódy Get() a Set() sú optimalizované pre rýchly prístup a zápis. Vlastnosti a metódy obrazových iterátorov spomenuté vyššie využijeme na jednoduchý príklad. Úlohou je vypočítať druhú mocninu všetkých hodnôt vo vstupnom obraze a zapísať ich do výstupného obrazu.

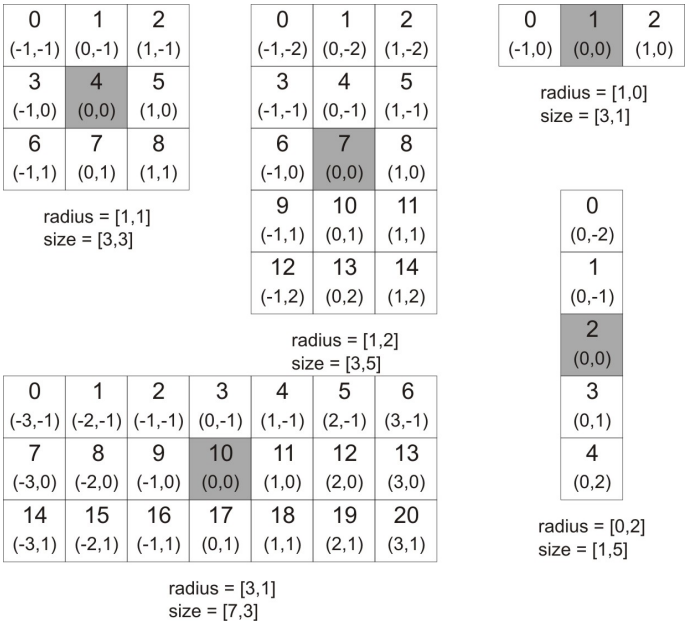
```
ConstIteratorType
    in( inputImage, inputImage->GetRequestedRegion() );
IteratorType
    out( outputImage, outputImage->GetRequestedRegion() );
for ( in.GoToBegin(), out.GoToBegin();
  !in.IsAtEnd(); ++in, ++out )
    {
    out.Set( in.Get() * in.Get() );
    }
```

Krása tejto ukážky kódu je v tom, že platí rovnako pre 1,2 i 3-rozmerné dáta a žiadna vedomosť o rozmeroch obrazu nie je potrebná.

1.4 Obrazový iterátor s okolím

Pri množstve algoritmov na spracovanie obrazu je potrebné poznať okolie aktuálne spracovávaného pixelu, čo znamená, že výsledok je vypočítaný z hodnôt pixelov v N-rozmernom okolí. Ako príklad môže slúžiť výpočet gradientu (derivácie).

Táto sekcia opisuje triedu ITK obrazových iterátorov, ktoré sú navrhnuté na prácu s pixelmi a ich okoliami. ITK iterátor s okolím prechádza oblasť tak ako normálny obrazový iterátor, ale namiesto odkazovania len na jeden pixel v každom okamihu, odkazuje na celé okolie pixelu. Rozšírené rozhranie poskytuje prístup pre čítanie a zápis ku všetkým pixelom v okolí.



Obr. 1.2: Zobrazenie rôznych tvarov okolia pixelu

V ITK je okolie pixelu definované ako malá množina pixelov okolo daného pixelu. Veľkosť a tvar okolia sa môže meniť v závislosti od aplikácie.

Iterátor s okolím používa tie isté operátory ako sú definované v predchádzajúcej kapitole a tie isté konštrukcie kódu ako normálny iterátor. Oproti štandardnému obrazovému iterátoru, konštruktor iterátora s okolím vyžaduje okrem smerníka na obraz a iteračnej oblasti ďalší argument, ktorý špecifikuje rozmer okolia. Okolie pixelu je symetrické podľa jeho stredného pixelu a je dané ako pole N celočíselných kladných hodnôt reprezentujúce jednotlivé polomery. Spolu sa nazývajú rádius. Na obrázka (Obr. 1.2) je zobrazený vzťah medzi rádiusom iterátora a veľkosťou okolia pre rôzne 2D tvary iterátora s okolím. Metódy poskytujúce informácie o okolí iterátora sú:

- SizeType GetRadius() vracia N-rozmerný rádius okolia ako objekt triedy itk::Size.
- unsigned long Size() vracia N-rozmerné pole celých čísel, ktoré predstavujú rozmery okolia.

Jeden spôsob ako si predstaviť okolie pixelu je 1-rozmerné pole, kde každý pixel má svoj unikátny celočíselný index. Na obrázku (Obr. 1.2) je jedinečný celočíslený index zobrazený vo vrchnej časti každého pixelu. Stredný pixel je vždy na pozícii n/2, kde n je veľkosť poľa.

- GetPixel(const unsigned int i) vráti hodnotu pixelu z okolia na pozícii i.
- void SetPixel(const unsigned int i, PixelType p) nastaví hodnotu pixelu z okolia na pozícii i na hodnotu p.

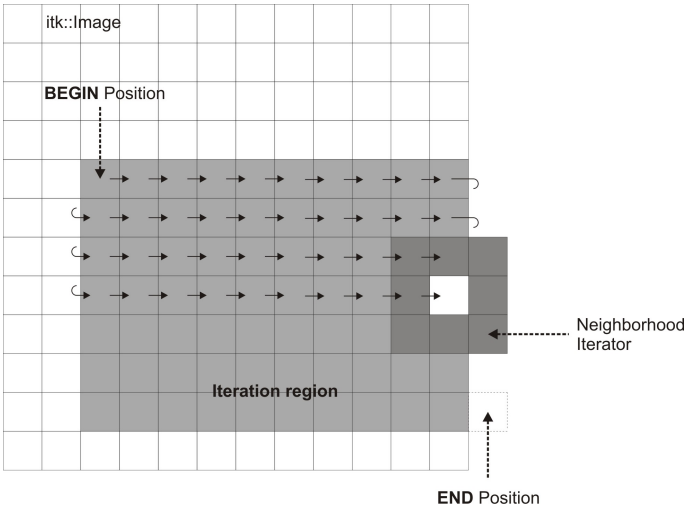
Druhý spôsob je udávať pozíciu pixelu v okolí pomocou posunutia od stredného pixelu. Ľavý horný roh z 3x3 okolia môžeme opísať pomocou posunu (-1,-1), kým pravý dolný roh toho istého okolia bude mať posun rovný (1,1). Na obrázku (Obr. 1.2) je posun od stredného pixelu zobrazený v spodnej časti každého pixelu.

- PixelType GetPixel(const OffsetType &o) vráti hodnotu pixelu na pozícii zodpovedajúcej posunu o od stredu okolia.
- void SetPixel(const OffsetType &, PixelType p) nastaví hodnotu pixelu na pozícii zodpovedajúcej posunu o od stredu okolia na hodnotu p.

Taktiež existujú špeciálne metódy na načítanie a zápis hodnôt do stredného pixelu okolia

- PixelType GetCenterPixel() vráti hodnotu pixelu v strede okolia.
- void SetCenterPixel(PixelType p) nastaví hodnotu v strede okolia na hodnotu p.

Výpočty založené na hodnotách pixelov z okolia vyžadujú v blízkosti hranice údaje, ktoré spadajú mimo oblasti. Pre iterátor na obrázku (Obr.1.3) so stredným pixelom na hranici pixelu, traja susedia z jeho okolia v obraze neexistujú. Ak časť okolia pixelu spadá mimo obraz, hodnoty pre chýbajúce pixely sú doplnené podľa pravidla daného matematickým modelom riešeného problému. Toto pravidlo sa nazýva okrajová podmienka. ITK iterátory s okolím automaticky detekujú odkazovanie na pixely mimo oblasti a vracajú hodnoty podľa okrajovej podmienky. Typ okrajovej podmienky je špecifikovaný pomocou druhého, voliteľného parametra šablóny iterátora. Štandardne je použitá Neumannova podmienka, pri ktorej skalárny súčin gradientu a normály k hranici je na hranici oblasti rovný nule. V ITK sú preddefinované ďalšie okrajové podmienky, napríklad Dirichletove alebo periodické.



Obr. 1.3: Zobrazenie iteračnej oblasti a iterátora s okolím

Kapitola 2

Matematické modely

Zmena je život. V prenesenom zmysle slova sa dá zmena fyzikálnej veličiny alebo funkčnej hodnoty vyjadriť pomocou derivácie. Derivácia je silný nástroj umožňujúci zapísať zmeny v pochopiteľnej a prehľadnej forme. Všetky zmeny nenastávajú len v jednom smere, ale sú plné rozmanitostí. Ďalším základným nástrojom prírody je rovnosť. Spracovaním všetkých predchádzajúcich pojmov do jedného výrazu dostávame parciálne diferenciálne rovnice (PDR). Aj v tejto práci sa pracuje s PDR. Nič však nie je také jednoduché ako na prvý pohľad vyzerá, preto sa do týchto krásnych pravidiel a zákonov dostáva nelinearita. Nelinearita nie je problém, skôr prekážka. Prekážka sa dá obísť rôznymi spôsobmi. V ďalšom popíšem spôsob, akým sme sa vyhli prekážke menom nelinearita na jednej nelineárnej PDR. Tou PDR je tzv. Mean Curvature Flow (MCF). Okrem tejto rovnice budeme v ďalšom hovoriť aj o rovnici Perona-Malik (PM), Slowed Mean Curvature Flow (SMCF) a Geodesic Mean Curvature Flow (GMCF). Ak sa nám už podarí odstrániť nelinearitu, sme stále len v polčase. Čaka nás riešenie, v tomto prípade už lineárneho systému rovníc.

2.1 Semi-implicitná schéma PDR

Nájsť klasické riešenie PDR v analytickom tvare je pre väčšinu PDR takmer nemožné. V tejto oblasti sú nám nápomocné výpočtové technológie. Vlastnosťou tohto prístupu je nutnosť delenia výpočtovej oblasti na menšie časti a výpočet hodnôt neznámej funkcie len v diskrétnych bodoch tejto oblasti. Ako z názvu vyplýva, na uskutočnenie cieľa z predchádzajúcich riadkov využijeme semi-implicitnú schému. Princíp tohto prístupu [2],[3] vysvetlím na

príklade rovnice Mean Curvature Flow (MCF). Ide o rovnicu v tvare:

$$\partial_t u - |\nabla u| \nabla \cdot \left(\frac{\nabla u}{|\nabla u|} \right) = 0, \qquad u(t,x) : [0,T] \times \Omega \to R \tag{2.1}$$

kde $\Omega \subset R^d$, pričom $d = 2$ alebo $d = 3$ je priestorová oblasť, na ktorej riešime rovnicu. Neznáma funkcia u závisí aj od času t, ktorý nadobúda hodnotu z intervalu $[0, T]$, kde T je daný koncový čas. Okrajová podmienka rovnice je

$$\frac{\partial u}{\partial \nu} = 0, \qquad na \;\; \partial\Omega \tag{2.2}$$

kde ν je jednotková vonkajšia normála ku hranici oblasti $\partial\Omega$. Keďže rovnicu riešime aj v časovej oblasti, potrebujeme začiatočnú podmienku

$$u(0,x) = u^0(x), \qquad pre \;\; x \in \Omega \tag{2.3}$$

kde $u^0(x)$ je daný obraz, na ktorý chceme aplikovať rovnicu MCF.

Táto rovnica vyjadruje pohyb izočiar (izoplôch) funkcie u v smere normály rýchlosťou, ktorá je daná ich strednou krivosťou. Model sa dá použiť na filtráciu (odšumenie) obrazu vďaka zhladzovaciemu efektu, ktorý je dôsledkom pohybu izočiar (ide o sťahovanie izočiar).

Ako prvý krok urobíme diskretizáciu časovej derivácie. Z predchádzajúceho máme dané, že $t \in [0, T]$. Tento interval si rozdelíme na n_t rovnako dlhých časových krokov, kde dĺžku časového kroku označíme τ. Už v tomto kroku sa stavia základ celej metódy. Hodnotu numerického riešenia v časovom kroku n označíme horným indexom u^n.

Prvý člen na ľavej strane rovnice (2.1), deriváciu neznámej podľa času, aproximujeme pomocou konečnej diferencie. V druhom člene ľavej strany rovnice (2.1) budeme brať nelineárny člen - absolútnu hodnotu gradientu neznámej ($|\nabla u|$) z predchádzajúceho časového kroku $(n-1)$, kým lineárny člen - samotný gradient neznámej (∇u) vezmeme z aktuálneho časového kroku (n). Po týchto úpravách bude mať rovnica nasledujúci tvar:

$$\frac{u^n - u^{n-1}}{\tau} - |\nabla u^{n-1}| \nabla \cdot \left(\frac{\nabla u^n}{|\nabla u^{n-1}|} \right) = 0 \tag{2.4}$$

Naša neznáma nie je len funkciou času, ale aj priestoru, preto treba vykonať aj diskretizáciu v priestore. V tomto prípade využijeme metódu konečných objemov. Celú oblasť Ω si rozdelíme na menšie časti, pričom zabezpečíme, aby jeden objem zodpovedal práve jednému obrazovému bodu (pixel, voxel). Pre označenie konečných objemov použijeme dolný index. Aktuálny konečný objem označíme indexom p (napr. u_p^n), susedné konečné objemy indexom

q (napr. u_q^n). Všetky členy predchádzajúcej rovnice (2.4) zintegrujeme cez všeobecný konečný objem V_p, z čoho dostávame:

$$\int_{V_p} \frac{u^n - u^{n-1}}{\tau} d\Omega - \int_{V_p} |\nabla u^{n-1}| \nabla \cdot \left(\frac{\nabla u^n}{|\nabla u^{n-1}|} \right) d\Omega = 0 \tag{2.5}$$

Nasledujúca úprava spočíva v tom, že prvý člen ľavej strany rovnice (2.5) aproximujeme pomocou súčinu miery (plochy, objemu) konečného objemu ($m(V_p)$) a hodnoty argumentu integrálu v ťažisku konečného objemu (u_p^n, príp. u_p^{n-1}).

Ďalej aproximujeme aj druhý člen ľavej strany rovnice (2.5). Veľkosť gradientu neznámej na konečnom objeme počítame z hodnôt u, ktoré sú v čase výpočtu známe. Ak označíme priemernú hodnotu tejto veličiny v objeme V_p ako $\bar{Q}_p^{n-1}$, môžeme tento člen vybrať pred integrál, čím rovnica získa tvar:

$$m(V_p) \frac{u_p^n - u_p^{n-1}}{\tau} - \bar{Q}_p^{n-1} \int_{V_p} \nabla \cdot \left(\frac{\nabla u^n}{|\nabla u^{n-1}|} \right) d\Omega = 0 \tag{2.6}$$

Integrál cez konečný objem v rovnici (2.6) môžeme aplikáciou Greenovej vety o divergencii preniesť na integrál cez hranicu konečného objemu (∂V_p), pričom sa zmení aj argument integrálu a upravený tvar rovnice bude vyzerať:

$$m(V_p) \frac{u_p^n - u_p^{n-1}}{\tau} - \bar{Q}_p^{n-1} \int_{\partial V_p} \frac{\nabla u^n \cdot \nu}{|\nabla u^{n-1}|} d\sigma = 0 \tag{2.7}$$

kde ν označuje jednotkovú normálu k hranici konečného objemu. Keďže obrazový bod je pravouhlá oblasť (obdĺžnik, resp. kváder), môžeme hranicu konečného objemu vidieť ako množinu úsečiek, resp. obdĺžnikov. Integrál cez celú hranicu konečného objemu sa zmení na sumu integrálov po častiach hranice (stranách obdĺžnika, resp. stranách kvádra), súhrne označených ako e_{pq}.

$$m(V_p) \frac{u_p^n - u_p^{n-1}}{\tau} - \bar{Q}_p^{n-1} \sum_{q \in N(p)} \int_{e_{pq}} \frac{\nabla u^n \cdot \nu}{|\nabla u^{n-1}|} d\sigma = 0 \tag{2.8}$$

kde množina $N(p)$ predstavuje indexy q všetkých konečných objemov, ktoré majú s konečným objemom p spoločnú hranicu nenulovej dĺžky.

Na integrál cez časť hranice e_{pq} použijeme znovu podobnú aproximáciu ako pri aproximácii prvého člena rovnice (2.5). Tento integrál nahradíme súčinom miery (dĺžka, obsah) časti hranice ($m(e_{pq})$) a priemernej hodnoty argumentu integrálu na tejto časti hranice, ktorý označíme indexom (pq). Dostávame:

$$m(V_p) \frac{u_p^n - u_p^{n-1}}{\tau} - \bar{Q}_p^{n-1} \sum_{q \in N(p)} m(e_{pq}) \frac{\nabla u_{pq}^n \cdot \nu_{pq}}{|\nabla u_{pq}^{n-1}|} = 0 \tag{2.9}$$

Skalárny súčin gradientu neznámej ∇u_{pq}^n a vonkajšej normály ν_{pq} v strede časti hranice aproximujeme za predpokladu, že sieť pixelov sa nachádza v pravouhlom súradnicovom systéme použitím centrálnej diferencie ako podiel rozdielu hodnôt neznámej v strede susedného a aktuálneho obrazového bodu $(u_q^n - u_p^n)$ a ich vzájomnej vzdialenosti $m(pq)$. Predchádzajúci tvar sa zmení na:

$$m(V_p)\frac{u_p^n - u_p^{n-1}}{\tau} - \bar{Q}_p^{n-1} \sum_{q \in N(p)} m(e_{pq}) \frac{u_q^n - u_p^n}{m(pq)} \frac{1}{|\nabla u_{pq}^{n-1}|} = 0 \tag{2.10}$$

Jediné neznáme hodnoty sú u_p^n a u_q^n. Ak chceme pretvoriť náš tvar rovnice na tvar vhodný pre výpočet systému lineárnych rovníc nasledovného tvaru:

$$a_p u_p^n - \sum_{q \in N(p)} a_{pq} u_q^n = b_p \tag{2.11}$$

musíme za jednotlivé koeficienty a_p, a_{pq} a b_p dosadiť nasledovné:

$$b_p = \frac{m(V_p)}{\tau} u_p^{n-1} \tag{2.12}$$

$$a_{pq} = \bar{Q}_p^{n-1} \frac{m(e_{pq})}{m(pq)|\nabla u_{pq}^{n-1}|} \tag{2.13}$$

$$a_p = \frac{m(V_p)}{\tau} + \sum_{q \in N(p)} a_{pq} = \frac{m(V_p)}{\tau} u_p^{n-1} + \sum_{q \in N(p)} \bar{Q}_p^{n-1} \frac{m(e_{pq})}{m(pq)|\nabla u_{pq}^{n-1}|} \tag{2.14}$$

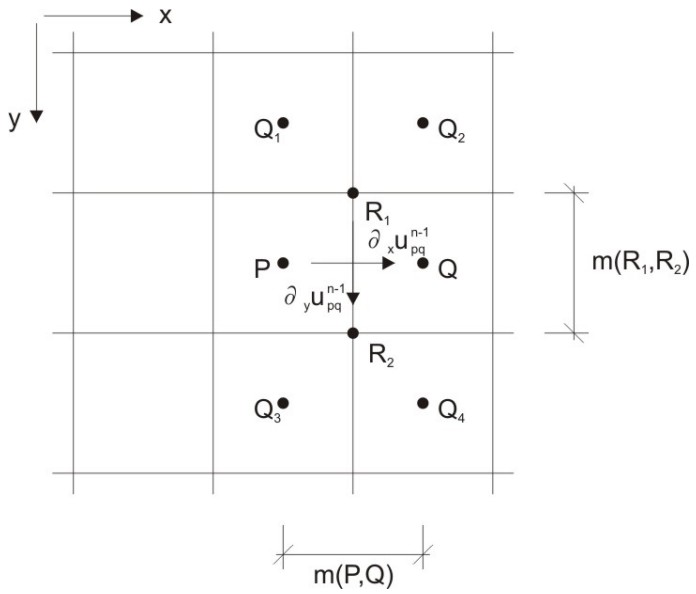
Posledná vec pred úspešným riešením spomenutého lineárneho systému je zistiť hodnoty výrazov $\bar{Q}_p^{n-1}$ a $|\nabla u_{pq}^{n-1}|$. Veľkosť gradientu v strede obrazového bodu $\bar{Q}_p^{n-1}$ aproximujeme ako priemer aproximovaných veľkostí gradientov v stredoch jednotlivých častí hranice $|\nabla u_{pq}^{n-1}|$.

$$\bar{Q}_p^{n-1} = \frac{1}{m(N(p))} \sum_{q \in N(p)} |\nabla u_{pq}^{n-1}| \tag{2.15}$$

kde $m(N(p))$ predstavuje kardinalitu množiny $N(p)$.

Potrebuje ešte vypočítať veľkosti gradientov neznámej na jednotlivých častiach hranice ($|\nabla u_{pq}^{n-1}|$). Výpočet vysvetlíme na dvojrozmernom prípade na obrázku (Obr.2.1). Na získanie hodnoty gradientu na hranici medzi konečnými objemami P a Q potrebujeme poznať parciálne derivácie v jednotlivých súradnicových smeroch.

$$|\nabla u_{pq}^{n-1}| = \sqrt{(\partial_x u_{pq}^{n-1})^2 + (\partial_y u_{pq}^{n-1})^2} \tag{2.16}$$



Obr. 2.1: Výpočet gradient

Parciálne derivácie nahradíme konečnými diferenciami, konkrétne centrálnou diferenciou. Pre smer kolmý na hranicu nemáme problém, pretože poznáme hodnoty, ktoré využívame pri výpočte. Pre hranu medzi konečnými objemami P a Q na obrázku (Obr. 2.1) je to smer x a dostávame

$$\partial_x u_{pq}^{n-1} = \frac{u_P - u_Q}{m(P,Q)} \tag{2.17}$$

Pre druhý smer, z príkladu na obrázku (Obr. 2.1) mu zodpovedá smer y a platí podobný vzťah daný tvarom:

$$\partial_y u_{pq}^{n-1} = \frac{u_{R_1} - u_{R_2}}{m(R_1,R_2)} \tag{2.18}$$

Problémom je, že hodnoty neznámej v týchto bodoch nepoznáme. Dopočítame ich ako priemer hodnôt v stredoch susedných pixelov, v ktorých hodnoty poznáme.

$$u_{R_1} = \frac{1}{4}(u_P + u_Q + u_{Q_1} + u_{Q_2}) \tag{2.19}$$

$$u_{R_2} = \frac{1}{4}(u_P + u_Q + u_{Q_3} + u_{Q_4}) \tag{2.20}$$

Dosadením (2.19) a (2.20) do (2.18) si vystačíme s jednoduchým vzorcom na výpočet absolútnej hodnoty gradientu (2.16). Podobným spôsobom je možné vypočítať hodnoty aj na ďalších hranách konečného objemu.

Predtým ako použijeme predchádzajúci výpočet absolútnej hodnoty gradientu musíme si uvedomiť, že táto veličina môže nadobúdať nulovú hodnotu. Tento prípad spôsobí problém pri výpočte a to z dôvodu nuly v menovateli v člene s veľkosťou gradientu neznámej (2.13) a (2.14). Túto komplikáciu vyriešime regularizáciou v nasledujúcom tvare

$$|\nabla u_{pq}^{n-1}|_\epsilon = \sqrt{|\nabla u_{pq}^{n-1}|^2 + \epsilon^2} \tag{2.21}$$

kde ϵ je v našom prípade zvolená veľmi malá konštanta ($\epsilon << 1$).

Toto bola posledná úprava potrebná na úspešné zostavenie systému lineárnych rovníc.

2.2 Riešenie lineárneho systému

Lineárny systém rovníc musíme ešte vyriešiť. Pri výpočtoch na počítačoch sa vo veľkej miere využívajú iteračné metódy. Medzi tieto metódy patrí napr. Jacobiho, Gauss-Seidelova, alebo nami používaná metóda succesive over-relaxation (SOR). Metóda SOR je rozšírením Gauss-Seidelovej metódy.

Majme štvorcový lineárny systém n rovníc s neznámou $\mathbf{x}$

$$A\mathbf{x} = \mathbf{b} \tag{2.22}$$

kde

$$A = \begin{pmatrix} a_{11} & a_{12}\dots & a_{1n} \\ a_{21} & a_{22}\dots & a_{2n} \\ \vdots & \vdots\ddots & \vdots \\ a_{n1} & a_{n2}\dots & a_{nn} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{2.23}$$

Iteračný proces začína zvolením štartovacieho vektora $\mathbf{x^0}$. Pre výpočet novej iterácie $\mathbf{x^{k+1}}$, $k = 0, 1\dots$ platí

$$x_i^{k+1} = (1-\omega)x_i^k + \frac{\omega}{a_{ii}}\left(b_i - \sum_{i>j} a_{ij}x_j^k - \sum_{i<j} a_{ij}x_j^{k+1}\right), \quad i = 1, 2, , \dots, n \tag{2.24}$$

kde ω je relaxačný parameter a nutná podmienka konvergencie je, aby $0 < \omega < 2$.

V našich príkladoch nie je matica A plná a namiesto indexov i a j používame označenie konečných objemov p a q. Taktiež namiesto neznámej funkcie

x používame označenie u. Začiatočný odhad u_p^0 v časovom kroku n numerickej metódy je v našom prípade rovný hodnote u_p z časového kroku $n-1$. Tak dostaneme tvar

$$u_p^{k+1} = (1-\omega)u_p^k + \frac{\omega}{a_p}\left(b_p - \sum_{q\in N_1(p)} a_{pq}u_{pq}^k - \sum_{q\in N_2(p)} a_{pq}u_{pq}^{k+1}\right), \quad \forall p \in \Omega \tag{2.25}$$

kde $N_1(p)$ je množina indexov konečných objemov z okolia konečného objemu p, z ktorých zatiaľ nepoznáme hodnotu neznámej funkcie u v novej iterácii. Množina $N_2(p)$ predstavuje naopak indexy konečných objemov, v ktorých poznáme hodnotu neznámej funkcie u v novej iterácii.

2.3 Explicitná a implicitná schéma

Je zrejmé, že semi-implicitná schéma nie je jediná možnosť, ktorou možno riešiť danú rovnicu (2.1). Znovu z názvu vyplýva, že sa budeme venovať dvom ďalším alternatívam: explicitnej a implicitnej schéme. Jediný, avšak zásadný rozdiel spočíva hneď v prvom kroku úpravy rovnice, časovej diskretizácii.

Pre zopakovanie, semi-implicitná schéma využíva konečnú diferenciu na aproximáciu časovej derivácie. Gradient neznámej sa počíta v aktuálnom časovom kroku (∇u^n) a veľkosti gradientov neznámej sa počítajú v predchádzajúcom časovom kroku ($|\nabla u^{n-1}|$).

2.3.1 Explicitná schéma

Naproti tomu sa pri explicitnej schéme získavajú hodnoty gradientu ale aj veľkosti gradientu z hodnôt u v predchádzajúcom časovom kroku. Výsledná schéma má tvar:

$$\frac{u^n - u^{n-1}}{\tau} - |\nabla u^{n-1}|\nabla \cdot \left(\frac{\nabla u^{n-1}}{|\nabla u^{n-1}|}\right) = 0 \tag{2.26}$$

Pomocou rovnakých úprav ako pri semi-implicitnej schéme dostávame výsledný tvar rovnice:

$$m(V_p)\frac{u_p^n - u_p^{n-1}}{\tau} - \bar{Q}_p^{n-1}\sum_{q\in N(p)} m(e_{pq})\frac{u_q^{n-1} - u_p^{n-1}}{m(pq)}\frac{1}{|\nabla u_{pq}^{n-1}|} = 0 \tag{2.27}$$

V tejto rovnici je len jedna neznáma u_p^n. Z tohto dôvodu nie je potrebné riešiť žiaden lineárny systém rovníc a priamo dostávame hodnotu v aktuálnom časovom kroku.

2.3.2 Implicitná schéma

Podobným spôsobom odvodíme aj implicitnú schému. Pre implicitnú schému platí, že hodnoty gradientu neznámej a veľkosti gradientu neznámej počítame z hodnôt neznámej v aktuálnom časovom kroku. Tvar rovnice sa zmení na:

$$\frac{u^n - u^{n-1}}{\tau} - |\nabla u^n| \nabla \cdot \left(\frac{\nabla u^n}{|\nabla u^n|} \right) = 0 \tag{2.28}$$

Tento malý rozdiel medzi semi-implicitnou a implicitnou schémou spôsobuje, že by sme sa nedopracovali k lineárnemu systému rovníc. Dostali by sme ťažko riešiteľný nelineárny systém rovníc. Táto nelinearita je dôsledkom výpočtu veľkosti gradientu $|\nabla u^n_{pq}|$ pomocou druhej odmocniny.

2.3.3 Zhodnotenie

Explicitná schéma, pri ktorej je možné počítať hodnoty neznámej funkcie v novom časovom kroku priamo z hodnôt v predchádzajúcom časovom kroku nám zabezpečuje vysokú rýchlosť výpočtu. Túto výhodu obmedzuje nutnosť použitia malého časového kroku pre zachovanie stability výpočtu.

Výhodou implicitnej schémy je jej bezpodmienečná stabilita vzhľadom na dĺžku časového kroku. Za túto výhodu však zaplatíme časovo náročným výpočtom nelineárneho systému rovníc, pričom je navyše často potrebný veľmi presný začiatočný odhad riešenia.

Nakoniec, semi-implicitná schéma spája výhody oboch predchádzajúcich. Z explicitnej schémy si berie rýchlosť výpočtu, ktorý síce nie je taký rýchly ako pri explicitnej schéme, avšak výrazne rýchlejší ako pri implicitnej schéme. Od implicitnej schémy si berie výhodu bezpodmienečnej stability výpočtu. Získavame tak optimálny nástroj na numerické riešenie spomenutého matematického modelu.

2.4 Ďalšie modely na filtráciu obrazu

Ako bolo spomenuté na začiatku kapitoly, okrem MCF rovnice budeme pracovať aj s ďalšími matematickými modelmi na filtráciu obrazu. Schémy pre tieto modely už nebudeme odvádzať tak podrobne ako schému pre prvý model. Zameriame sa len na výsledné tvary diskretizačných schém.

Rovnica Perona-Malik (PM) má tvar:

$$\partial_t u - \nabla \cdot (g(|\nabla u|)\nabla u) = 0, \qquad u(t,x) : [0,T] \times \Omega \to R \tag{2.29}$$

Rovnica Slowed Mean Curvature Flow (SMCF) má tvar:

$$\partial_t u - g(|\nabla u|)|\nabla u|\nabla \cdot \left(\frac{\nabla u}{|\nabla u|}\right) = 0, \qquad u(t,x) : [0,T] \times \Omega \to R \tag{2.30}$$

Rovnica Geodesic Mean Curvature Flow (GMCF) [4] má tvar:

$$\partial_t u - |\nabla u|\nabla \cdot \left(g(|\nabla u|)\frac{\nabla u}{|\nabla u|}\right) = 0, \qquad u(t,x) : [0,T] \times \Omega \to R \tag{2.31}$$

kde funkcia g je tzv. detektor hrán a je v každom modeli definovaná nasledovne

$$g(s) = \frac{1}{1 + Ks^2}, \; K > 0 \tag{2.32}$$

Pre všetky tieto rovnice platia rovnaké okrajové podmienky (2.2) a začiatočná podmienka (2.3) ako pri rovnici MCF (2.1).

Po vykonaní časovej semi-implicitnej diskretizácii pre jednotlivé modely dostávame.

PM:

$$\frac{u^n - u^{n-1}}{\tau} - \nabla \cdot \left(g(|\nabla u^{n-1}|)\nabla u^n\right) = 0 \tag{2.33}$$

SMCF:

$$\frac{u^n - u^{n-1}}{\tau} - g(|\nabla u^{n-1}|)|\nabla u^{n-1}|\nabla \cdot \left(\frac{\nabla u^n}{|\nabla u^{n-1}|}\right) = 0 \tag{2.34}$$

GMCF:

$$\frac{u^n - u^{n-1}}{\tau} - |\nabla u^{n-1}|\nabla \cdot \left(g(|\nabla u^{n-1}|)\frac{\nabla u^n}{|\nabla u^{n-1}|}\right) = 0 \tag{2.35}$$

Ďalším krokom je priestorová diskretizácia pomocou metódy konečných objemov. Obdobnými úpravami ako v prípade rovnice MCF získame výsledný tvar diskretizačnej schémy.

PM:

$$m(V_p)\frac{u_p^n - u_p^{n-1}}{\tau} - \sum_{q \in N(p)} m(e_{pq})\frac{u_q^n - u_p^n}{m(pq)} g(|\nabla u_{pq}^{n-1}|) = 0 \tag{2.36}$$

SMCF:

$$m(V_p)\frac{u_p^n - u_p^{n-1}}{\tau} - g(\bar{Q}_p^{n-1})\bar{Q}_p^{n-1} \sum_{q \in N(p)} m(e_{pq})\frac{u_q^n - u_p^n}{m(pq)}\frac{1}{|\nabla u_{pq}^{n-1}|} = 0 \tag{2.37}$$

GMCF:

$$m(V_p)\frac{u_p^n - u_p^{n-1}}{\tau} - \bar{Q}_p^{n-1} \sum_{q \in N(p)} m(e_{pq}) \frac{u_q^n - u_p^n}{m(pq)} \frac{1}{|\nabla u_{pq}^{n-1}|} g(|\nabla u_{pq}^{n-1}|) = 0 \quad (2.38)$$

Z predchádzajúceho tvaru je možné získať koeficienty do lineárneho systému. Jednotlivé koeficienty majú nasledujúci tvar.

PM:

$$b_p = \frac{m(V_p)}{\tau} u_p^{n-1} \quad (2.39)$$

$$a_{pq} = \frac{m(e_{pq})}{m(pq)} g(|\nabla u_{pq}^{n-1}|) \quad (2.40)$$

$$a_p = \frac{m(V_p)}{\tau} + \sum_{q \in N(p)} a_{pq} = \frac{m(V_p)}{\tau} + \sum_{q \in N(p)} \frac{m(e_{pq})}{m(pq)} g(|\nabla u_{pq}^{n-1}|) \quad (2.41)$$

SMCF:

$$b_p = \frac{m(V_p)}{\tau} u_p^{n-1} \quad (2.42)$$

$$a_{pq} = g(\bar{Q}_p^{n-1}) \bar{Q}_p^{n-1} \frac{m(e_{pq})}{m(pq)|\nabla u_{pq}^{n-1}|} \quad (2.43)$$

$$a_p = \frac{m(V_p)}{\tau} + \sum_{q \in N(p)} a_{pq} = \frac{m(V_p)}{\tau} + \sum_{q \in N(p)} g(\bar{Q}_p^{n-1}) \bar{Q}_p^{n-1} \frac{m(e_{pq})}{m(pq)|\nabla u_{pq}^{n-1}|} \quad (2.44)$$

GMCF:

$$b_p = \frac{m(V_p)}{\tau} u_p^{n-1} \quad (2.45)$$

$$a_{pq} = \bar{Q}_p^{n-1} \frac{m(e_{pq})}{m(pq)|\nabla u_{pq}^{n-1}|} g(|\nabla u_{pq}^{n-1}|) \quad (2.46)$$

$$a_p = \frac{m(V_p)}{\tau} + \sum_{q \in N(p)} a_{pq} = \frac{m(V_p)}{\tau} + \sum_{q \in N(p)} \bar{Q}_p^{n-1} \frac{m(e_{pq})}{m(pq)|\nabla u_{pq}^{n-1}|} g(|\nabla u_{pq}^{n-1}|) \quad (2.47)$$

2.5 Semi-implicitná metóda pre segmentáciu obrazu

Princíp semi-implicitnej metódy pre časovú diskretizáciu, metódy konečných objemov pre priestorovú diskretizáciu a riešenia lineárneho systému pomocou SOR nie je obmedzený len na matematické modely popisujúce filtráciu

obrazu. V tejto časti práce aplikujeme predchádzajúci postup na jednu z rovníc slúžiacich na segmentáciu obrazu. Týmto matematickým modelom je tzv. Generalized Subjective Surface Model (GSUBSURF) [5], [6].

Pri segmentácii pomocou GSUBSURF potrebujeme dva vstupné obrazy. Prvý u^g predstavuje obraz, ktorý chceme segmentovať. Z tohto obrazu si na samotnom začiatku vypočítame hodnoty, ktoré využijeme ďalej (detektor hrán). Druhým obrazom je u^0, začiatočná podmienka, ktorú v čase vyvíjame pomocou rovnice GSUBSURF.

Rovnica GSUBSURF má nasledujúci tvar:

$$\partial_t u - w_{con}\ \nabla g\ \cdot \nabla u - w_{dif} g|\nabla u| \nabla \cdot \left(\frac{\nabla u}{|\nabla u|} \right) = 0 \quad u(t,x) : [0,T] \times \Omega \to R \tag{2.48}$$

Taktiež platia rovnaké okrajové podmienky (2.2) a začiatočná podmienka (2.3) ako v prípade rovnice MCF.

Rovnako ako v predchádzajúcich odvodeniach, funkcia g má nasledujúci tvar

$$g(s) = \frac{1}{1 + Ks^2},\ K > 0 \tag{2.49}$$

Malým, ale dôležitým rozdielom je, že do argumentu funkcie g budú vstupovať len hodnoty z $|\nabla u^g|$.

Použitie koeficientov w_{con} a w_{dif} možno považovať za váhy advekcie a difúzie v modeli. Týmito parametrami získavame manipulačný priestor na lepšie korigovanie priebehu procesu segmentácie.

Pri odvádzaní tvaru rovnice, ktorý je vhodný na riešenie lineárneho systému pomocou SOR sme sa inšpirovali prácou [7]. Prvým krokom je podobne ako v predchádzajúcich prípadoch semi-implicitná časová diskretizácia s časovým krokom τ. Prvý člen rovnice (2.48), časovú deriváciu, aproximujeme konečnou diferenciou. Gradient funkcie u v druhom člene ľavej strany rovnice budeme brať z predchádzajúceho časového kroku. V treťom člene vypočítame absolútne hodnoty gradientu funkcie u z predchádzajúceho časového kroku, kým gradient funkcie u bez absolútnej hodnoty vypočítame z aktuálneho časového kroku.

$$\frac{u^n - u^{n-1}}{\tau} - w_{con} \nabla g\ \cdot \nabla u^{n-1} - w_{dif} g\ |\nabla u^{n-1}| \nabla \cdot \left(\frac{\nabla u^n}{|\nabla u^{n-1}|} \right) = 0 \tag{2.50}$$

Nasleduje integrácia cez celý konečný objem V_p:

$$\int_{V_p} \frac{u^n - u^{n-1}}{\tau} d\Omega - \int_{V_p} w_{con} \nabla g \cdot \nabla u^{n-1} d\Omega - \int_{V_p} w_{dif} g |\nabla u^{n-1}| \nabla \cdot \left(\frac{\nabla u^n}{|\nabla u^{n-1}|} \right) d\Omega = 0 \tag{2.51}$$

Prvý člen ľavej strany predchádzajúcej rovnice aproximujeme pomocou súčinu mierou konečného objemu (plocha, objem) a diferencie hodnôt zo stredov konečných objemov.

$$m(V_p)\frac{u_p^n - u_p^{n-1}}{\tau} \tag{2.52}$$

Na úpravu druhého člena rovnice (2.51) využijeme pravidlo v nasledujúcom tvare

$$\mathbf{a} \cdot \nabla b = \nabla \cdot (\mathbf{a}b) - b\nabla \cdot \mathbf{a} \tag{2.53}$$

Dosadením $(-w_{con}\nabla g)$ za $\mathbf{a}$ a (u^{n-1}) za b získame

$$\int_{V_p} \nabla \cdot (-w_{con}\nabla g \; u^{n-1})d\Omega - \int_{V_p} u^{n-1}\nabla \cdot (-w_{con}\nabla g)d\Omega \tag{2.54}$$

Následne z druhej časti pravej strany rovnice (2.54) vyberieme pred integrál hodnotu u^{n-1}, ktorú aproximujeme hodnotou v strede konečného objemu u_p^{n-1}. Ďalej využitím Greenovej vety o divergencii prevedieme objemový integrál cez konečný objem na integrál cez hranicu konečného objemu. Tento integrál bude reprezentovaný ako suma cez časti hranice obrazového bodu (strana, stena) aktuálneho konečného objemu a susedných objemov. Po aplikácii dostávame

$$\sum_{q\in N(p)} \int_{e_{pq}} (-u^{n-1}w_{con}\nabla g \cdot \nu)d\sigma - u_p^{n-1} \sum_{q\in N(p)} \int_{e_{pq}} -w_{con}\nabla g \cdot \nu d\sigma \tag{2.55}$$

Ďalej budeme rozlišovať hranice na výtokové (outflow) a prítokové (inflow) a to definovaním dvoch množín indexov, kde $N^{in}(p)$ je množina susedných konečných objemov q, pre ktoré platí $(-w_{com}\nabla g \cdot \nu) \leq 0$ a $N^{out}(p)$ je množina susedných konečných objemov q, pre ktoré platí $(-w_{com}\nabla g \cdot \nu) > 0$. Ak použijeme princíp upwind pre aproximáciu prvého člena výrazu (2.55) dostaneme

$$\sum_{q\in N^{out}(p)} \int_{e_{pq}} -u_p^{n-1}w_{con}\nabla g \cdot \nu d\sigma + \sum_{q\in N^{in}(p)} \int_{e_{pq}} -u_q^{n-1}w_{con}\nabla g \cdot \nu d\sigma \tag{2.56}$$

Dosadením (2.56) do (2.55) a po úprave získame výsledný tvar druhého člena rovnice (2.51)

$$\sum_{q\in N^{in}(p)} \int_{e_{pq}} (u_q^{n-1} - u_p^{n-1})(-w_{con}\nabla g \cdot \nu)d\sigma \tag{2.57}$$

Pri aproximácii tretieho člena rovnice (2.51) použijeme podobné prístupy ako pri filtračných algoritmoch. Člen w_{dif} je na celom konečnom objeme

konštantný a preto ho môžeme presunúť pred integrál. Funkciu g nahradíme konštantou g_p, ktorá bude predstavovať priemernú hodnotu funkcie g na celom konečnom objeme. Za tohto predpokladu ju môžeme rovnako presunúť pred integrál. Obdobným spôsobom nahradíme aj člen absolútnej hodnoty gradientu funkcie u za $\bar{Q}_p^{n-1}$, ktorý tiež predstavuje priemernú hodnotu danej veličiny na konečnom objeme. Po týchto úpravách dostávame

$$\int_{V_p} w_{dif} g |\nabla u^{n-1}| \nabla \cdot \left(\frac{\nabla u^n}{|\nabla u^{n-1}|} \right) d\Omega = w_{dif} g_p \bar{Q}_p^{n-1} \int_{V_p} \nabla \cdot \left(\frac{\nabla u^n}{|\nabla u^{n-1}|} \right) d\Omega \tag{2.58}$$

Opätovne aplikáciou Greenovej vety o divergencii, nahradením integrálu cez hranicu oblasti sumou integrálov cez časti hranice medzi konečným objemom a jeho susedmi dostaneme výsledný tvar tretieho člena ľavej strany rovnice (2.51)

$$w_{dif} g_p \bar{Q}_p^{n-1} \sum_{q \in N(p)} \int_{e_{pq}} \left(\frac{\nabla u^n \cdot \nu}{|\nabla u^{n-1}|} \right) d\sigma \tag{2.59}$$

Posledným krokom, ktorý nás delí od získania konečnej diskretizačnej schémy, je náhrada skalárneho súčinu normály k hranici konečného objemu (ν) a gradientu funkcie u (člen 2.59), príp. g (člen 2.57) ako v prípade modelov určených na filtráciu použitím diferencie.

Spojením jednotlivých diskretizovaných častí (2.52), (2.57) a (2.59) dohromady získame diskretizačnú schému v tvare:

$$\begin{aligned} m(V_p)\tfrac{u_p^n - u_p^{n-1}}{\tau} - \textstyle\sum_{q \in N^{in}(p)} (u_p^{n-1} - u_q^{n-1}) \left(-w_{com} m(e_{pq}) \tfrac{g_q - g_p}{m(gp)} \right) \\ -w_{dif} g_p \bar{Q}_p^{n-1} \textstyle\sum_{q \in N(p)} m(e_{pq}) \tfrac{u_q^n - u_p^n}{m(pq)|\nabla u_{pq}^{n-1}|} = 0 \end{aligned} \tag{2.60}$$

Z tejto schémy dokážeme vyjadriť všetky členy potrebné do lineárneho systému. Tie budú nasledujúceho tvaru:

$$b_p = m(V_p)\frac{u_p^{n-1}}{\tau} + \sum_{q \in N^{in}(p)} (u_p^{n-1} - u_q^{n-1})(-w_{com} m(e_{pq}) \frac{g_q - g_p}{m(gp)}) \tag{2.61}$$

$$a_{pq} = w_{dif} g_p \bar{Q}_p^{n-1} m(e_{pq}) \frac{1}{m(pq)|\nabla u_{pq}^{n-1}|} \tag{2.62}$$

$$a_p = \frac{m(V_p)}{\tau} + \sum_{q \in N(p)} a_{pq} = \frac{m(V_p)}{\tau} + w_{dif} g_p \bar{Q}_p^{n-1} \sum_{q \in N(p)} m(e_{pq}) \frac{1}{m(pq)|\nabla u_{pq}^{n-1}|} \tag{2.63}$$

Kapitola 3

Implementácia v ITK

Po predchádzajúcich dvoch kapitolách o ITK a matematických modeloch nám nezostáva nič iné ako nazbierané vedomosti aplikovať a vytvoriť niečo nové. Pri implementácii vyššie spomenutých semi-imlicitných metód na filtráciu a segmentáciu obrazu snažili sme sa v čo najväčšej miere dodržiavať a využívať princípy, na ktorých je založené ITK.

Knižnice ITK obsahujú už množstvo filtrov na odšumenie a segmentáciu obrazu. Filtre založené na numerickom riešení PDR však využívajú len explicitnú časovú diskretizáciu. Väčšina z nich je odvodená od triedy itk::FiniteDifferenceImageToImageFilter. Úprava tejto triedy na semi-implicitnú metódu nie je vôbec priamočiara, skôr ťažkopádna.

Na základe týchto poznatkov sme sa rozhodli vytvoriť novú triedu SemiImplicitFilter ako potomka triedy itk::ImageToImageFilter. Táto trieda predstavuje základnú triedu pre filtre, pri ktorých je ako vstup i výstup obraz.

Pri tvorbe tejto triedy bolo naším základným cieľom vytvoriť triedu, ktorá by poskytovala všeobecný podklad pre implementáciu konečno-diferenčných a konečno-objemových schém, ktoré vedú na riešenie lineárneho systému. Inými slovami, ide o metódy, pri ktorých je hodnota numerického riešenia v danom uzlovom bode (v strede pixelu, voxelu) vyjadrená ako lineárna kombinácia hodnôt v okolitých uzlových bodoch. Veľkosť použitého okolia a voľba týchto uzlových bodov nie je triedou obmedzená, tieto údaje nastavuje sám programátor pri implementácii svojho filtra.

Ako bolo spomenuté v kapitole o ITK, kód triedy sa rozdeľuje do dvoch súborov, v tomto prípade to budú SemiImplicitFilter.h a SemiImplicitFilter.txx.

3.1 SemiImplicitFilter.h

Zo začiatku kapitoly je zrejmé, že naša trieda SemiImpicitImageToImageFilter je odvodená od triedy ImageToImageFilter. Z toho dôvodu je prvoradé vloženie zodpovedajúceho hlavičkového súboru.

```
#include "itkImageToImageFilter.h"
```

a následne zadefinovať samotné odvodenie tried

```
template <class TInputImage, class TOutputImage>
class SemiImpicitImageToImageFilter
 : public ImageToImageFilter<TInputImage, TOutputImage>
```

V ITK je štandardom, že pre lepšiu čitateľnosť kódu sa použité dátové typy označujú skrátenými názvami.

Základné definície tried:

```
typedef SemiImplicitFilter Self;
typedef
    ImageToImageFilter<TInputImage, TOutputImage> Superclass;
typedef SmartPointer<Self>       Pointer;
typedef SmartPointer<const Self> ConstPointer;
```

Vstupné a výstupné typy obrazu

```
typedef typename Superclass::InputImageType  InputImageType;
typedef typename Superclass::OutputImageType OutputImageType;
typedef typename OutputImageType::Pointer    OutputImagePointer;
```

Typ pixelu, na ktorom budú prebiehať výpočty je rovnakého typu ako výstupný pixel

```
typedef typename
    Superclass::OutputImageType::PixelType  PixelType;
```

Ďalej nasleduje množstvo vstavaných funkcii triedy itk::ImageToImageFilter, z ktorých má pre nás najväčší význam funkcia GenerateData() v ktorej prebieha samotný výpočet.

```
virtual void GenerateData();
```

V ďalšom texte rozpíšeme nami doplnené funkcie a premenné slúžiace na implementáciu základného tvaru semi-impicitných metód.

3.1.1 Premenné

- int m_MYNTAU - počet časových krokov
- int m_MYNITER - maximálny počet iterácií pre metódu SOR
- int m_MYRADIUS - veľkosť okolia aktuálneho obrazového bodu, z ktorého budeme získavať hodnoty pre výpočet
- double m_MYTAU - dĺžka časového kroku
- double m_MYOMEGA - parameter ω pri výpočte SOR
- double m_MYTHRESHOLD - požadovaná presnosť pri výpočte SOR

To boli premenné štandardných dátových typov C++. Nasledujú dátové typy vytvorené samotným ITK. V tomto prípade pre uchovávanie hodnôt v aktuálnom časovom kroku, aktuálnej a predchádzajúcej iterácie metódy SOR ako aj koeficientov a_p, a_{pq} a b_p nevyužívame klasické polia, ale pomôžeme si prácou, vykonanou vývojármi ITK a namiesto polí využijeme ich dátový typ itk::Image. Ak zoberieme priamo dátový typ reprezentujúci výstupný obraz, uľahčíme si prácu, pretože nemusíme riešiť dimenzionalitu, rozmer polí, či ich dátový typ a preto využijeme:

- OutputImagePointer uu - obraz reprezentujúci aktuálny časový krok
- OutputImagePointer uuold - obraz reprezentujúci predchádzajúcu (starú) iteráciu metódy SOR
- OutputImagePointer uunew - obraz reprezentujúci aktuálnu (novú) iteráciu metódy SOR
- OutputImagePointer bp - obraz reprezentujúci pravú stranu lineárneho systému
- OutputImagePointer *apq - smerník na pole obrazov reprezentujúce koeficienty lineárného systému

V predchádzajúcej kapitole o matematických modeloch sme používali osobitné označenie pre koeficienty a_p a a_{pq}. Pri implementácii sme však koeficient a_p zahrnuli do poľa apq, kde predstavuje strednú pozíciu v rámci okolia.

V samotnom kóde to vyzerá nasledovne:

```
int m_MYNTAU;
int m_MYNITER;
int m_MYRADIUS;
double m_MYTAU;
double m_MYOMEGA;
double m_MYTHRESHOLD;
OutputImagePointer uu;
OutputImagePointer uuold;
OutputImagePointer uunew;
OutputImagePointer bp;
OutputImagePointer *apq;
```

3.1.2 Funkcie

Okrem premenných sme dopracovali aj potrebné funkcie:

- void Initialization() - slúži na alokáciu jednotlivých polí (obrazov) použitých pri výpočte
- void Coefficients() - funkcia, v ktorej prebieha výpočet koeficientov lineárneho systému
- void SORSolver() - riešič lineárneho systému pomocou SOR
- void ReadU() - slúži na načítanie vstupného obrazu do obrazu uu
- void WriteU() - zapisuje obraz uu do výstupného obrazu
- void CopyUtoUNEW() (void CopyUNEWtoU(), void CopyUNEWtoUOLD()) - ako z ich názvov vyplýva,slúžia na kopírovanie hodnôt z jednej premennej do druhej
- double Norm() funkcia na výpočet normy rozdielu obrazov z dvoch posledných iterácii

Okrem týchto „výkonných“ funkcií sú potrebné aj nasledujúce „nastavovacie“ funkcie na nastavenie jednotlivých parametrov, ako napríklad void SetMYNTAU(int a).

Znovu ukážka kódu

```
void Initialization();
void Coefficients();
void SORSolver();
void ReadU();
```

```
  void WriteU();
  void CopyUtoUNEW();
  void CopyUNEWtoU();
  void CopyUNEWtoUOLD();
  double Norm();
  void SetMYNTAU(int a);
  void SetMYNITER(int a);
  void SetMYRADIUS(int a);
  void SetMYTAU(double a);
  void SetMYOMEGA(double a);
  void SetMYTHRESHOLD(double a);
```

3.2 SemiImplicitFilter.txx

Tak sa dostávame k druhej časti kódu. Tou je súbor .txx, v ktorom je rozpracovaný kód programu, ktorého štruktúru „nadiktoval“ súbor .h.

3.2.1 GenerateData

Ako sme v predchádzajúcom spomenuli vyššie najdôležitejšou funkciou tejto triedy je GenerateData(). Na začiatku tejto funkcie sa nachádzajú základné inicializačné funkcie ITK na alokáciu výstupov. Túto funkcionalitu zabezpečujú nasledujúce dve funkcie

```
this->AllocateOutputs();
this->CopyInputToOutput();
```

Na začiatok si definujeme pomocné premenné int t pre aktuálny časový krok, int iter pre aktuálnu iteráciu a double norma pre aktuálnu normu rozdielu novej a starej iterácie, ktorá sa porovnáva so zastavujúcim kritériom m_MYTHRESHOLD.

```
int t;
int iter;
double norm;
```

Ďalšiu časť predstavujú naše inicializačné funkcie, ktorých popis zaznel vyššie, a to konkrétne Initialization(), ReadU(), CopyUtoUNEW().

```
Initialization();
ReadU();
CopyUtoUNEW();
```

Nasleduje začiatok cyklu for cez časové kroky.

```
for(t=0;t<m_MYNTAU;t++){/*BEGIN FOR*/
```

V každom časovom kroku sa vypočítajú nové koeficienty lineárneho systému pomocou metódy Coefficients(). Premenná iter sa vynuluje, kým premenná norm sa nastaví napríklad na dvojnásobok hodnoty m_MYTHRESHOLD.

```
Coefficients();
iter=0;
norm=2*m_MYTHRESHOLD;
```

Ďalším krokom je cyklus while pre iteračný proces výpočtu SOR. Tento cyklus prebieha, kým sa nedosiahne maximálny počet iterácii m_MYNITER alebo norma dvoch predchádzajúcich obrazov norm nie je menšia ako hodnota m_MYTHRESHOLD.

```
while((iter<m_MYNITER)&&(norma>m_MYTHRESHOLD)){/*BEGIN WHILE*/
```

Jedna slučka iterácie pozostáva zo štyroch krokov. Prvým krokom je skopírovanie obrazu novej iterácie do starej pomocou metódy CopyUNEWtoUOLD(). Potom sa vypočíta riešenie systému SOR metódou SORSolver(). Nasleduje výpočet a zápis novej normy (norm=Norm()). Na záver cyklu while je nutné inkrementovať premennú iter.

```
CopyUNEWtoUOLD();
SORSolver();
norm=Norm();
iter++;
/*END WHILE*/}
```

Po ukončení cyklu while je nutné ešte pred ukončením cyklu for zapísať novú iteráciu do starého časového kroku, aby mohol začať výpočet nového časového kroku.

```
CopyUNEWtoU();
/*END FOR*/}
```

Na koniec funkcie GenerateData() je ešte potrebné vyvolať funkciu WriteU(), ktorá zapíše obraz posledného časového kroku do výstupného obrazu.

```
WriteU();
/*END GenerateData*/}
```

3.2.2 Initialization

Táto funkcia slúži na alokovanie pamäte pre jednotlivé polia (obrazy) zodpovedajúce časovým krokom, iteráciam alebo koeficientom do riešiča SOR. Ako príklad uvediem alokáciu jedného obrazu uu. Celý proces sa skladá z troch krokov. Po prvé pomocou metódy New() priradíme do premennej uu smerník na obraz zodpovedajúceho typu. Ako druhý krok nastavíme oblasť, s ktorou sa bude ďalej pracovať. Pre našu potrebu budeme využívať celý výstupný obraz, preto využijeme metódu GetOutput->GetRequestedRegion(), ktorá vráti oblasť veľkosti výsledného obrazu. Tretím a posledným krokom je samotná alokácia pomocou metódy Allocate().

```
uu=OutputImageType::New();
uu->SetRegions(this->GetOutput()->GetRequestedRegion());
uu->Allocate();
```

Rovnaký postup platí pre všetky ostatné premenné dátového typu OutputImageType (uuold, uunew, bp). Jedna malá zmena nastane pri premennej apq, ktorá je smerníkom na daný dátový typ, pretože predstavuje pole dátového typu OutputImageType. Najprv si zadefinujeme premennú n, ktorá bude predstavovať veľkosť okolia. Následne v cykle alokujeme pre jednotlivé prvky poľa apq obraz, ako je popísané vyššie pre uu.

```
int i;
int n=pow(2*m_MYRADIUS+1,OutputImageType::ImageDimension);
apq=new OutputImagePointer[n];
for(i=0;i<n;i++)
    {
    apq[i]=OutputImageType::New();
    apq[i]->SetRegions(this->GetOutput()->GetRequestedRegion());
    apq[i]->Allocate();
    }
```

3.2.3 Coefficients

Vo funkcii Coefficients() prebieha výpočet koeficientov lineárneho systému. Týmito koeficientmi sú apq a bp. V základnej triede SemiImplicitFilter sú koeficienty nastavené tak, aby nemenili vstupný obraz. To znamená, že koeficienty apq budú nulové pre všetky indexy okrem stredného pixelu (index rovný n/2), kde n je veľkosť okolia. Pre tento index bude koeficient nadobúdať hodnotu 1. Koeficient bp bude rovný hodnote v strednom pixeli, ktorú získame metódou GetCenterPixel(). Toto rozdelenie spôsobí, že počas celého priebehu sa hodnoty pixelov nebudú meniť.

Najprv si zadefinujeme typy pre jednotlivé iterátory nasledovne:

```
typedef itk::ConstNeighborhoodIterator< InputImageType >
    NeighborhoodIteratorType;
typedef itk::ImageRegionIterator< InputImageType > IteratorType;
```

Týmto krokom sme zadefinovali okrajové podmienky, ktoré sú zahrnuté v NeighborhoodIteratorType a to tým, že sme použili len jeden parameter šablóny. Takto sme nastavili okrajové podmienky na Neumannove, ktoré sú v ITK ako predvolené. Ak by sme chceli použiť iný typ okrajových podmienok, museli by sme k parametru InputImageType pridať druhý, dobrovoľný, ktorým by sme nastavili iné okrajové podmienky.

Pre iterátory cez okolie musíme ešte zadefinovať a nastaviť rádius tohto okolia:

```
NeighborhoodIteratorType::RadiusType radius;
radius.Fill(m_MYRADIUS);
```

Ďalším krokom je inicializácia jednotlivých iterátorov. Na obraze, ktorý zodpovedá aktuálnemu časového kroku uu použijeme iterátor s okolím iteruu, kým pre obraz pravej strany bp a koeficienty apq použijeme obyčajný obrazový iterátor (iterbp, iter). Pre iterátory na obrazy koeficientov apq (iter[i]) musíme použiť podobný trik so smerníkom ako na samotné obrazy.

```
NeighborhoodIteratorType
    iteruu(radius, uu,uu->GetRequestedRegion() );
IteratorType iterbp(bpp, bpp->GetRequestedRegion());
IteratorType *iter;
int i;
int n=pow(2*m_MYRADIUS+1,OutputImageType::ImageDimension);
iter=new IteratorType[n]);
for(i=0;i<n;i++)
    {
    iter[i]=IteratorType(apq[i], apq[i]->GetRequestedRegion());
    }
```

Teraz už môžeme nastaviť jednotlivé koeficienty. Najprv musíme všetky iterátory nastaviť na začiatok oblasti. Pre iterátory obrazov koeficientov apq (iter[i]) to vykonáme v samostatnom cykle, kým pre ostatné iterátory to môžeme zabezpečiť v hlavnom cykle.

```
for(i=0;i<n;i++) iter[i].GoToBegin();
for(iteruu.GoToBegin(),iterbp.GoToBegin();
    !iteruu.IsAtEnd();++iteruu,++iterbp){/*BEGIN FOR*/
```

V hlavnom cykle môžeme priamo nastaviť hodnotu pravej strany bp. Ostatné koeficienty vypočítame vo vnútornom cykle cez všetky body okolia apq. Ak je aktuálny index rovný n/2 (stredný pixel), koeficient sa nastaví na 1.0, kým pre ostatné indexy sa nastaví na 0.0.

```
iterbpp.Set(iteruu.GetCenterPixel());
for(i=0;i<n;i++)
    {
    if(i==n/2) {iter[i].Set(1.0);}
    else {iter[i].Set(0.0);
    }
```

Pred koncom hlavného cyklu for musíme ešte inkrementovať všetky iterátory obrazov koeficientov.

```
for(i=0;i<n;i++) ++iter[i];
/*END FOR*/}
```

3.2.4 SORSolver

V tejto funkcii prebieha celý výpočet novej iterácie pomocou metódy SOR. Rovnakým spôsobom ako v predchádzajúcej časti zadefinujeme a inicializujeme jednotlivé iterátory. Okrem vyššie spomenutých pribudne ešte iterátor pre aktuálnu (iteruunew) a predchádzajúcu iteráciu (iteruuold).

Podobne ako v predchádzajúcej časti musíme najprv nastaviť všetky iterátory na počiatok oblasti a začať cyklus cez všetky obrazové body.

```
for(i=0;i<n;i++) iter[i].GoToBegin();
for (
  iteruu.GoToBegin(),iteruunew.GoToBegin(),iteruuold.GoToBegin(),
  iterbp.GoToBegin(); !iteruu.IsAtEnd();
  ++iteruu,++iteruunew,++iteruuold,++iterbp){/*BEGIN FOR*/
```

V rámci cyklu si vypočítame pomocnú premennú suma, ktorá predstavuje súčin zodpovedajúcej dvojice z koeficientov (iter[i].Get()) a pixelu okolia (iteruu.GetPixel(i)). Následne musíme odrátať hodnotu súčinu prislúchajúcu k strednému pixelu (index rovný n/2), príp. pomocou metódy GetCenterPixel().

```
double suma;
suma=0.0;
for(i=0;i<n;i++)
    {
```

```
    suma+=iter[i].Get()*iteruunew.GetPixel(i);
    }
suma-=iter[n/2].Get()*iteruunew.GetCenterPixel();
```

Ďalším krokom je samotný výpočet SOR podľa vzorca (2.25).

```
iteruunew.SetCenterPixel(iteruunew.GetCenterPixel()
    +m_MYOMEGA*((iterbp.Get()+suma)/(iter[n/2].Get())
    -iteruunew.GetCenterPixel()));
```

Rovnako ako v predchádzajúcom prípade musíme pred ukončením cyklu inkrementovať všetky iterátory obrazov koeficientov.

```
for(i=0;i<n;i++) ++iter[i];
/*END FOR*/}
```

3.2.5 Ostatné metódy

Metódy ReadU(),WriteU(), CopyUtoUNEW(), CopyUNEWtoU(), CopyUNEWtoUOLD(),Norm() nebudem bližšie rozpisovať.

3.3 Filtračné algoritmy

Pre každý matematický model odvodíme zvlášť ďalšiu triedu, napr. MCFSemiImplicitFilter a pod. Vďaka štruktúre triedy popísanej vyššie bude pre tieto filtračné algoritmy potrebné pozmeniť len jedinú funkciu triedy, Coefficients(). Samozrejme pribudne ešte zopár kozmetických úprav. Jednotlivé zmeny oproti pôvodnej triede budem vysvetľovať na príklade rovnice MCF a 2-rozmernom obraze.

Inicializačná časť funkcie bude rovnaká, zmena nastane len vo vnútri cyklu cez obrazové body, v ktorom sa rátajú jednotlivé koeficienty apq a pravá strana bp.

Ešte predtým si však zadefinujeme premenné double ng1,ng2,ng3,ng4, ktoré prestavujú veľkosť gradientu neznámej na jednotlivých častiach hranice, veľkosť gradientu funkcie na celom objeme double Q a konštanty h, ktorá predstavuje veľkosť strany pixelu. Pre naše použitie bude platiť h=1.0.

```
double ng1,ng2,ng3,ng4,Q;
double h=1.0;
```

Prvým krokom je výpočet premenných ng1, ng2, ng3, ng4 a Q pre daný obrazový bod. Tieto hodnoty získame podľa vzorcov (2.15) a (2.16).

Pre sprehľadnenie zápisu si zadefinujeme nasledovné premenné typu PixelType, kde napríklad premenná u0 predstavuje hodnotu z obrazu uu v pixeli okolia s indexom 0.

```
u0=iteruu.GetPixel(0);
u1=iteruu.GetPixel(1);
u2=iteruu.GetPixel(2);
u3=iteruu.GetPixel(3);
u4=iteruu.GetPixel(4);
u5=iteruu.GetPixel(5);
u6=iteruu.GetPixel(6);
u7=iteruu.GetPixel(7);
u8=iteruu.GetPixel(8);
```

Potom môžeme prejsť k samotnému výpočtu

```
ng1=sqrt(
    pow((u6+u7-u0-u1)/(4.0*h),2.0)+pow((u4-u3)/(h),2.0));
ng2=sqrt(
    pow((u4-u1)/(h),2.0)+pow((u2+u5-u0-u3)/(4.0*h),2.0));
ng3=sqrt(
    pow((u7-u4)/(h),2.0)+pow((u5+u8-u3-u6)/(4.0*h),2.0));
ng4=sqrt(
    pow((u7+u8-u1-u2)/(4.0*h),2.0)+pow((u5-u4)/(h),2.0));
Q=(ng1+ng2+ng3+ng4)/(4.0);
```

Ďalej vstúpime do cyklu cez jednotlivé koeficienty. V tomto prípade budú nulové už len koeficienty na pozícii 0,2,6,8. Nenulové koeficienty 1,3,5,7 budú naberať hodnoty pre apq zo vzorca (2.13). Rovnako aj pre index rovný 4 bude platiť vzorec pre ap zo vzorca (2.14) a pre pravú stranu bp vzorec (2.12).

```
iterbp.Set(u4/m_MYTAU);
for(i=0;i<n;i++)
    {
    if(i==0) iter[i].Set(0);
    if(i==2) iter[i].Set(0);
    if(i==6) iter[i].Set(0);
    if(i==8) iter[i].Set(0);

    if(i==1) iter[i].Set((Q)/(ng2+0.000000000001));
    if(i==3) iter[i].Set((Q)/(ng1+0.000000000001));
    if(i==5) iter[i].Set((Q)/(ng4+0.000000000001));
    if(i==7) iter[i].Set((Q)/(ng3+0.000000000001));
```

```
    if(i==4) iter[i].Set(iterbp.Get()
        +(Q)/(ng2+0.000000000001)+(Q)/(ng1+0.000000000001)
        +(Q)/(ng4+0.000000000001)+(Q)/(ng3+0.000000000001));
    }
```

Týmto je zadefinovaná trieda na výpočet Mean Curvature Flow rovnice pomocou semi-implicitnej metódy konečných objemov.

Pre ďalšie tri metódy (Perona-Malik (PM),Slowed Mean Curvature Flow (SMCF),Geodesic Mean Curvature Flow (GMCF)) budeme musieť doplniť do hlavičkového súboru .h medzi premenné triedy ešte double K. Pre túto premennú budeme musieť taktiež zadefinovať nastavovaciu funkciu void SetMYK(double a).

```
double m_MYK;
void SetMYK(double a);
```

Po týchto krokoch nastanú znovu rovnaké zmeny ako v prípade MCF, s tým rozdielom, že sa budú inak počítať koeficienty pri indexoch 1,3,5,7 a 4. Koeficient pravej strany bp ostane rovnaký.

PM podľa vzorcov (2.40) a (2.41)

```
if(i==1) iter[i].Set(1.0/(1+K*ng2*ng2));
if(i==3) iter[i].Set(1.0/(1+K*ng1*ng1));
if(i==5) iter[i].Set(1.0/(1+K*ng4*ng4));
if(i==7) iter[i].Set(1.0/(1+K*ng3*ng3));

if(i==4) iter[i].Set(iterbp.Get()
    +1.0/(1+K*ng2*ng2)+1.0/(1+K*ng1*ng1)
    +1.0/(1+K*ng4*ng4)+1.0/(1+K*ng3*ng3));
```

SMCF podľa vzorcov (2.43) a (2.44)

```
if(i==1) iter[i].Set((Q*1.0/(1+K*Q*Q))/(ng2+0.000000000001));
if(i==3) iter[i].Set((Q*1.0/(1+K*Q*Q))/(ng1+0.000000000001));
if(i==5) iter[i].Set((Q*1.0/(1+K*Q*Q))/(ng4+0.000000000001));
if(i==7) iter[i].Set((Q*1.0/(1+K*Q*Q))/(ng3+0.000000000001));

if(i==4) iter[i].Set(iterbp.Get()
    +(Q*1.0/(1+K*Q*Q))/(ng2+0.000000000001)
    +(Q*1.0/(1+K*Q*Q))/(ng1+0.000000000001)
    +(Q*1.0/(1+K*Q*Q))/(ng4+0.000000000001)
    +(Q*1.0/(1+K*Q*Q))/(ng3+0.000000000001));
```

GMCF podľa vzorcov (2.46) a (2.47)

```
if(i==1) iter[i].Set((Q*1.0/(1+K*ng2*ng2))/(ng2+0.000000000001));
if(i==3) iter[i].Set((Q*1.0/(1+K*ng1*ng1))/(ng1+0.000000000001));
if(i==5) iter[i].Set((Q*1.0/(1+K*ng4*ng4))/(ng4+0.000000000001));
if(i==7) iter[i].Set((Q*1.0/(1+K*ng3*ng3))/(ng3+0.000000000001));

if(i==4) iter[i].Set(iterbp.Get()
    +(Q*1.0/(1+K*ng2*ng2))/(ng2+0.000000000001)
    +(Q*1.0/(1+K*ng1*ng1))/(ng1+0.000000000001)
    +(Q*1.0/(1+K*ng4*ng4))/(ng4+0.000000000001)
    +(Q*1.0/(1+K*ng3*ng3))/(ng3+0.000000000001));
```

3.4 Segmentačný algoritmus

Proces odvodenia algoritmu pre segmentáciu GSUBSURF bude o niečo zložitejší ako v predchádzajúcich častiach.

V hlavičkovom súbore .h budeme musieť doplniť nasledujúcu funkciu EdgeDetector(). Úlohou tejto funkcie je načítať obraz, z ktorého budú vypočítané potrebné hodnoty pre správny chod algoritmu (detektor hrán). Okrem tejto funkcie musíme ešte pridať niekoľko premenných (double m_MYWDIF, m_MYWCON a m_MYK), ktoré prestavujú jednotlivé parametre metódy popísanej v kapitole o semi-implicitnej segmentácii. Samozrejme, ku každej premennej pridáme aj nastavovaciu funkciu, napr. SetMYWDIF(double a). Okrem týchto premenných potrebujeme ešte dve premenné typu OutputImageType obrg a g, kde obrg bude predstavovať samotný obraz, ktorý načítame zo súboru. Premenná g predstavuje pole hodnôt funkcie podľa vzorca (2.49).

```
void EdgeDetector();
void SetMYWDIF(double a);
void SetMYWCON(double a);
void SetMYK(double a);

double m_MYWDIF;
double m_MYWCON;
double m_MYK;

OutputImagePointer g,obrg;
```

Najprv si v skratke popíšeme tvar funkcie EdgeDetector(). Ako prvé si musíme načítať obraz zo súboru do premennej obrg.

```
typedef itk::ImageRegionIterator< InputImageType> IteratorType;
IteratorType iter1(reader->GetOutput(),
    reader->GetOutput()->GetRequestedRegion());
IteratorType iter2(obrg,
    obrg->GetRequestedRegion());

for (iter1.GoToBegin(),iter2.GoToBegin();
    !iter1.IsAtEnd(); ++iter1,++iter2)
    {
    iter2.Set(iter1.Get());
    }
```

Následne si zadefinujeme potrebné iterátory a premenné

```
typedef itk::ConstNeighborhoodIterator< InputImageType >
    NeighborhoodIteratorType;
NeighborhoodIteratorType::RadiusType radius;
radius.Fill(m_MYRADIUS);

NeighborhoodIteratorType iterobrg(radius,
    obrg,obrg->GetRequestedRegion() );
IteratorType iterg(g, g->GetRequestedRegion());

double ng1,ng2,ng3,ng4,Q;
double h=1.0;
```

Ešte pred samotným výpočtom si podobne ako pri filtračných algoritmoch zadefinujeme pomocné premenné na sprehľadnenie kódu. Napríklad premenná g0 bude predstavovať hodnotu iterobrg.GetPixel(0).

```
g0=iterobrg.GetPixel(0);
g1=iterobrg.GetPixel(1);
g2=iterobrg.GetPixel(2);
g3=iterobrg.GetPixel(3);
g4=iterobrg.GetPixel(4);
g5=iterobrg.GetPixel(5);
g6=iterobrg.GetPixel(6);
g7=iterobrg.GetPixel(7);
g8=iterobrg.GetPixel(8);
```

Ďalej si vypočítame pomocné premennné ng1, ng2, ng3, ng4 a Q rovnakým spôsobom ako v predchádzajúcej kapitole o filtračných algoritmoch. Následne zapíšeme hodnotu funkcie g s argumetnom rovným premennej Q podľa vzorca (2.49) do premennej g cez iterátor iterg.

```
for(iterobrg.GoToBegin(),iterg.GoToBegin();
    !iterobrg.IsAtEnd(); ++iterobrg,++iterg)
    {
    ng1=sqrt(
        pow((g6+g7-g0-g1)/(4.0*h),2.0)+pow((g4-g3)/(h),2.0));
    ng2=sqrt(
        pow((g4-g1)/(h),2.0)+pow((g2+g5-g0-g3)/(4.0*h),2.0));
    ng3=sqrt(
        pow((g7-g4)/(h),2.0)+pow((g5+g8-g3-g6)/(4.0*h),2.0));
    ng4=sqrt(
        pow((g7+g8-g1-g2)/(4.0*h),2.0)+pow((g5-g4)/(h),2.0));
    Q=(ng1+ng2+ng3+ng4)/(4.0);
    iterg.Set(1.0/(1.0+K*Q*Q));
}
```

Presuňme sa k dôležitej funkcii Coefficients(). Tu sa podobne ako v predchádzajúcich filtračných algoritmoch bude meniť len cyklus for cez obrazové body. Výpočet pomocných premenných ng1,ng2,ng3,ng4 a Q je rovnaký ako v predchádzajúcich prípadoch. Rovnako aj koeficienty s indexmi 0,2,6,8 budú vždy nulové. Zmena nastáva len v zostávajúcich indexoch. Pre členy apq (indexy 1,3,5,7) platí vzorec (2.62)

```
if(i==1) iter[i].Set(
    (m_MYWDIF*iterg.GetPixel(4)*Q)/(ng2+0.000000000001));
if(i==3) iter[i].Set(
    (m_MYWDIF*iterg.GetPixel(4)*Q)/(ng1+0.000000000001));
if(i==5) iter[i].Set(
    (m_MYWDIF*iterg.GetPixel(4)*Q)/(ng4+0.000000000001));
if(i==7) iter[i].Set(
    (m_MYWDIF*iterg.GetPixel(4)*Q)/(ng3+0.000000000001));
```

Pre index 4, koeficient ap vo vzroci (2.63) dostávame

```
if(i==4) iter[i].Set(1.0/m_MYTAU
    +(m_MYWDIF*iterg.GetPixel(4)*Q)/(ng2+0.000000000001)
    +(m_MYWDIF*iterg.GetPixel(4)*Q)/(ng1+0.000000000001)
    +(m_MYWDIF*iterg.GetPixel(4)*Q)/(ng4+0.000000000001)
    +(m_MYWDIF*iterg.GetPixel(4)*Q)/(ng3+0.000000000001));
```

Ako posledný uvedieme vzťah pre koeficient pola bp, vzorec (2.61)

```
iterbp.Set(itereruu.GetPixel(4)/m_MYTAU
    +vijk1*(iteruu.GetPixel(4)-iteruu.GetPixel(1))
```

```
    +vijk3*(iteruu.GetPixel(4)-iteruu.GetPixel(3))
    +vijk5*(iteruu.GetPixel(4)-iteruu.GetPixel(5))
    +vijk7*(iteruu.GetPixel(4)-iteruu.GetPixel(7)));
```

kde premenné vijk1, vijk3, vijk5, vijk7 sú dané vzťahmi

```
vijk1=-m_MYWCON*(iterg.GetPixel(1)-iterg.GetPixel(4));
vijk3=-m_MYWCON*(iterg.GetPixel(3)-iterg.GetPixel(4));
vijk5=-m_MYWCON*(iterg.GetPixel(5)-iterg.GetPixel(4));
vijk7=-m_MYWCON*(iterg.GetPixel(7)-iterg.GetPixel(4));
```

Toto boli všetky zmeny od základnej triedy SemiImplicitFilter

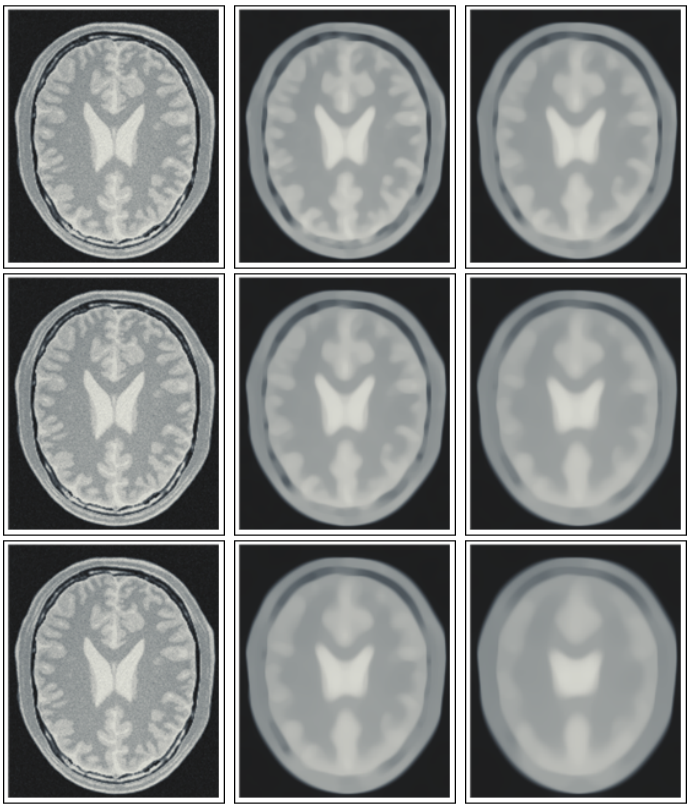
Kapitola 4

Obrazové výstupy

V tejto kapitole budú dominovať obrázky nad textom. Pri každej metóde je zobrazených niekoľko obrazových výstupov, pomocou ktorých demonštrujeme vplyv zmeny jednotlivých parametrov na výsledný obraz. Pod každým súborom obrazov je prehľadný popis týchto parametrov.

Z dôvodu lepšej prehľadnsti obrazov a textu, obrazové výstupy jednotlivých metód sa začínajú na novej strane.

4.1 Mean Curvature Flow



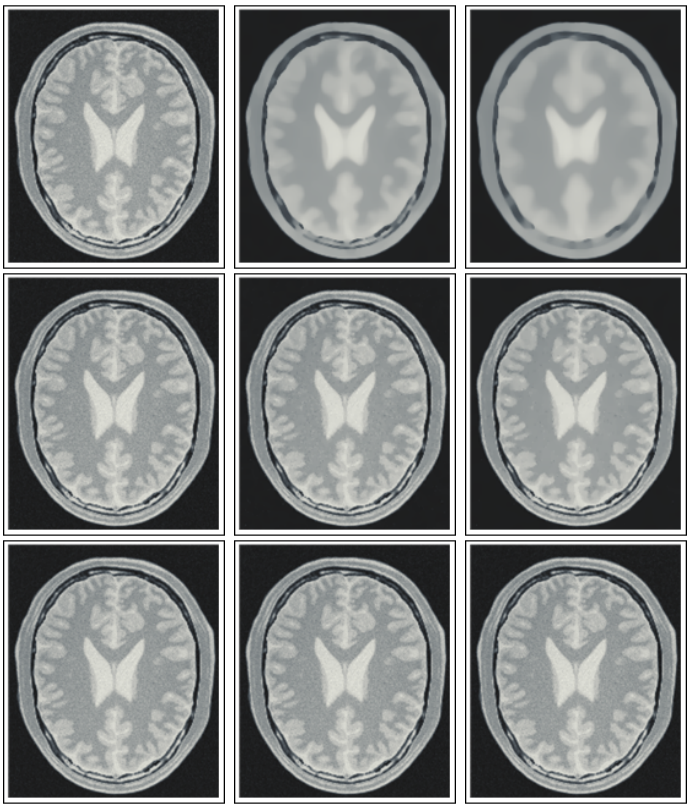
Obr. 4.1: Obrazové výstupy rovnice MCF

Obrázky v 1. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.25$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 6.25$ a $t = 12.5$).

Obrázky v 2. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

Obrázky v 3. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 1.0$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 25.0$ a $t = 50.0$).

4.2 Slowed Mean Curvature Flow



Obr. 4.2: Obrazové výstupy rovnice SMCF

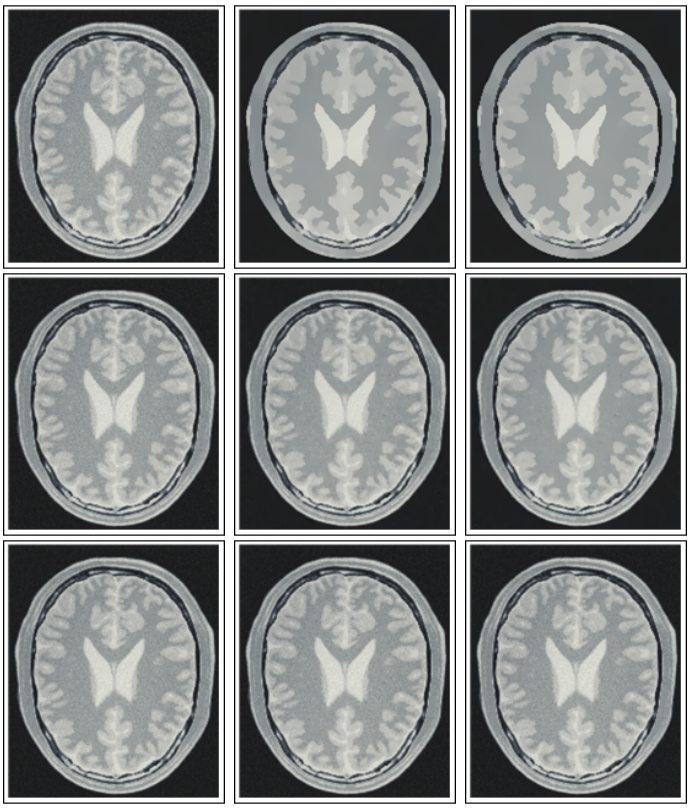
Obrázky v 1. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ a konštante $K = 0.01$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

Obrázky v 2. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ a konštante $K = 1.0$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

Obrázky v 3. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ a konštante $K = 100$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

Znižovaním konštanty K sa zväčšuje zhladzovací efekt rovnice SMCF.

4.3 Geodesic Mean Curvature Flow



Obr. 4.3: Obrazové výstupy rovnice GMCF

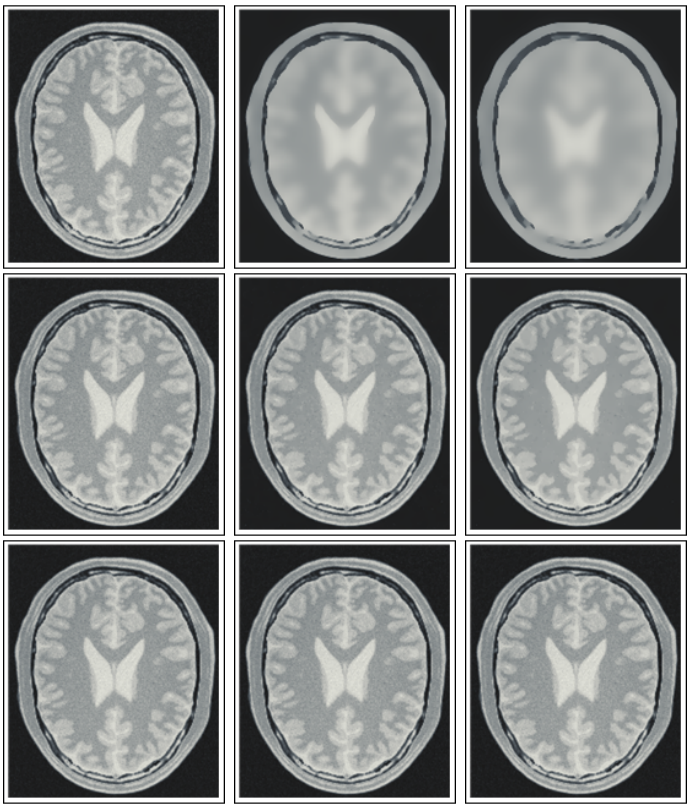
Obrázky v 1. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ a konštante $K = 0.01$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

Obrázky v 2. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ a konštante $K = 1.0$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

Obrázky v 3. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ a konštante $K = 100$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

Znižovaním konštanty K sa rovnako ako pri rovnici SMCF zväčšuje zhladzovací efekt.

4.4 Perona-Malik



Obr. 4.4:

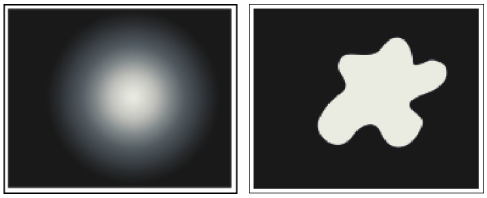
Obrázky v 1. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ a konštante $K = 0.01$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

Obrázky v 2. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ a konštante $K = 1.0$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

Obrázky v 3. riadku zobrazujú priebeh pre dĺžku časového kroku $\tau = 0.5$ a konštante $K = 100$ postupne (zľava doprava) pre 0., 25. a 50 časový krok, zodpovedajúce výslednému času ($t = 0$, $t = 12.5$ a $t = 25.0$).

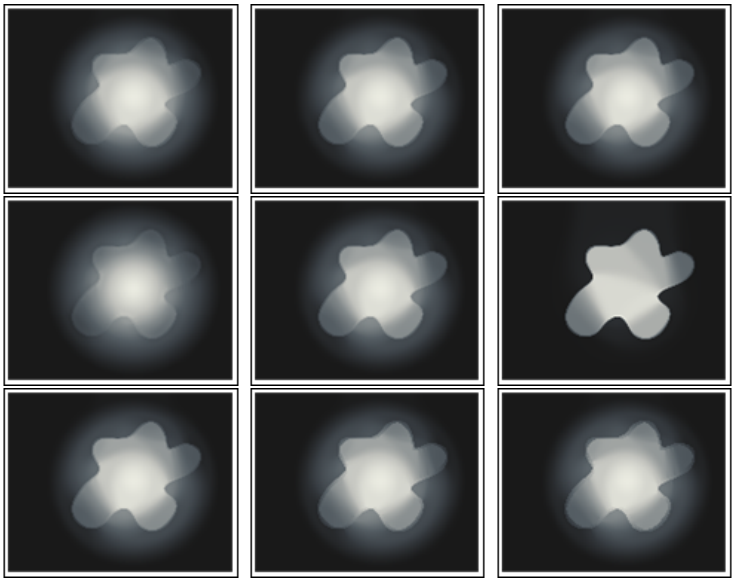
Znižovaním konštanty K sa rovnako ako v prípade predchádzajúcich dvoch rovníc zväčšuje zhladzovací.

4.5 Segmentácia GSUBSURF



Obr. 4.5: Vstupné obrazy pre GSUBSURF. Vľavo začiatočná podmienka u^0. Vpravo segmentovaný obraz u^g

.



Obr. 4.6: Obrazové výstupy rovnice GSUBSURF

Obrázky v 1. riadku zobrazujú výsledky pre koncový čas $t = 100$ s dĺžkou časového kroku $\tau = 1.0$, konštante $K = 1.0$ a konštante $w_{dif} = 1.0$. Postupne (zľava doprava) nadobúda konštanta w_{con} hodnoty (0.1, 0.5 a 1.0).

Obrázky v 2. riadku zobrazujú výsledky pre koncový čas $t = 100$ s dĺžkou časového kroku $\tau = 1.0$, konštante $K = 1.0$ a konštante $w_{con} = 1.0$. Postupne (zľava doprava) nadobúda konštanta w_{dif} hodnoty (0.1, 1.0 a 10).

Obrázky v 3. riadku zobrazujú výsledky pre koncový čas $t = 100$ s dĺžkou časového kroku $\tau = 1.0$, konštante $w_{con} = 1.0$ a konštante $w_{dif} = 1.0$. Postupne (zľava doprava) nadobúda konštanta K hodnoty (1, 100 a 10000).

Najväčší vplyv na rýchlosť segmentácie má zmena parametra w_{dif}.

Záver

Podarilo sa nám úspešne implementovať rôzne matematické modely na filtráciu a segmentáciu obrazu v duchu princípov balíka knižníc ITK. Dôležitou časťou tohto úspechu bolo naštudovanie a pochopenie systému práce s triedami, metódami a premennými ITK. Pri samotnej implementácii semi-implicitných metód sme sa rozhodli pre vytvorenie úplne novej triedy, ktorá môže slúžiť množstvu semi-implicitných metód. Týmto krokom sme dosiahli určitú voľnosť pri vytváraní triedy a v maximálnej možnej miere využívali výhody práce s ITK.

Zoznam použitej literatúry

[1] Ibanez L., Schroeder W., Cates J. et al.: ITK Software Guide, Kitware Inc. 2005

[2] Bourgine P., Cunderlik R., Drblikova O., Mikula K., Peyrieras N., Remesikova M., Rizzi B., Sarti A.: 4D embryogenesis image analysis using PDE methods of image processing, Kybernetika, Vol. 46, No. 2, 2010

[3] Krivá Z., Mikula K., Peyriéras N., Rizzi B., Sarti A.: Zebrafish early embryogenesis 3D image filtering by nonlinear partial differential equations. Medical Image Analysis. Submitted for publication.

[4] Caselles V., Kimmel R., Sapiro G.: Geodesic active contours, Internat J. Comput, Visio 22 1997, s. 61-79

[5] Zanella C., Campana M., Rizzi B., Melani C., Sanguinetti G., Bourgine P., Mikula K., Peyriéras N., Sarti A.: Cells segmentation from 3-D confocal images of early zebrafish embryogenesis, IEEE Trans. Image Process 19, 2010, s. 770-781

[6] Mikula K., Remešíková M.: Finite volume schemes for generalized subjective surface equation in image segmentation, Kybernetika 45, 2009

[7] Frolkovič P., Mikula K.: Flux-based level set method: A finite volume method for evolving interfaces, Appl. Numer. Math. 57 2007, s. 436-454