SIEMRI
MRI PROCESSING APP

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS,
COMPUTER SCIENCE AND BIOMEDICAL ENGINEERING

DEPARTMENT OF AUTOMATICS AND BIOMEDICAL ENGINEERING

DADM project documentation - draft

Authors:	*SIMENS Healthkare*
Degree programme:	*Biomedical engineering*
Supervisor:	*PhD. Tomasz Pięciak*

Kraków, 2017

Contents

1. **List of changes** 7

2. **Assumptions** 9

3. **Structure** 11

4. **User guide** 13

   4.1. Requirements 13

   4.2. Instruction 13

5. **Detailed description** 15

   5.1. Module 1. MRI reconstruction 15

   5.2. Module 2. Intensity inhomogeneity correction 17

   5.3. Module 3. Non-stationary noise estimation 18

   5.4. Module 4. Non-stationary noise filtering 1 19

   5.5. Module 5. Non-stationary noise filtering 2 20

   5.6. Module 6. Diffusion tensor imaging 22

      5.6.1. Weighted Least Squares estimation 23

      5.6.2. Nonlinear Least Squares estimation 24

      5.6.3. Diffusion Biomarkers 25

   5.7. Module 8. Skull stripping 26

   5.8. Module 9. Segmentation 27

   5.9. Module 10. Upsampling 29

   5.10. Module 11. Brain 3D 32

   5.11. Module 12. Oblique imaging 33

6. **Implementation** 35

   6.1. Tools 35

   6.2. Module 1. MRI reconstruction 35

   6.3. Module 2. Intensity inhomogenity correction 38

   6.4. Module 3. Non-stationary noise estimation 38

   6.5. Module 4. Non-stationary noise filtering 1 38

   6.6. Module 5. Non-stationary noise filtering 2 38

   6.7. Module 6. Diffusion tensor imaging 48

      6.7.1. Initialization 48

6.7.2. WLS estimation 49
6.7.3. NLS estimation 49
6.7.4. Biomarkers computation 51
6.7.5. Implementation caveats 54
6.8. Module 8. Skull stripping 54
6.9. Module 9. Segmentation 55
6.10. Module 10. Upsampling 55
6.11. Module 11. Brain 3D 57
6.12. Module 12. Oblique imaging 57
7. Tests 59
7.1. Module 1. MRI reconstruction 59
7.2. Module 2. Intensity inhomogenity correction 59
7.3. Module 3. Non-stationary noise estimation 59
7.4. Module 4. Non-stationary noise filtering 1 59
7.5. Module 5. Non-stationary noise filtering 2 59
7.6. Module 6. Diffusion tensor imaging 59
7.6.1. High-level tests 59
7.6.2. Low-level tests 60
7.7. Module 8. Skull stripping 61
7.8. Module 9. Segmentation 61
7.9. Module 10. Upsampling 61
7.10. Module 11. Brain 3D 65
7.11. Module 12. Oblique imaging 65
7.12. Application 65
8. Authors 67

1. List of changes

Name	Date	Details
Sylwia Mól	19-Nov-2017	Document created
Sylwia Mól	20-Nov-2017	Structure changed
Sylwia Mól	21-Nov-2017	Chapter "Authors" added, in-out table added
Malwina Molendowska	26-Nov-2017	Description of 1st module added
Eliza Kowalczyk	27-Nov-2017	Description of 9th module added
Karolina Gajewska	27-Nov-2017	Description of 11th module added
Eliza Kowalczyk	28-Nov-2017	Description of 10th module changed
Alicja Martinek	28-Nov-2017	Description of 5th module changed
Mateusz Pabian	29-Nov-2017	Description of 6th module changed
Jacek Fidos	29-Nov-2017	Tools description added
Anna Grzywa	29-Nov-2017	Description of 8th module added
Magdalena Rychlik	29-Nov-2017	Description of 4th module
Michał Kotarba	29-Nov-2017	Description of 12th module added
Magdalena Kucharska	29-Nov-2017	Description of 9th module added
Klaudia Gugulska	30-Nov-2017	Description of 2nd module added
Sylwia Mól	2-Dec-2017	Titles added, numeration changed
Jacek Fidos	3-Dec-2017	Files separation
Eliza Kowalczyk	9-Dec-2017	Enhancement of description of 10th module
Michał Kotarba	9-Dec-2017	Changed in-out for my module
Kacper Turek	9-Dec-2017	Description of 3rd module changed and in-out for my module added
Anna Grzywa	9-Dec-2017	In-out for 8th module added
Sylwia Mól	10-Dec-2017	"Structure" & "Assumptions" chapters changed
Klaudia Gugulska	10-Dec-2017	Update of description of 2nd module
Jacek Fidos	4-Jan-2018	Short SciPy description
Eliza Kowalczyk	14-Jan-2018	Upsampling documentation update
Sylwia Mól	17-Jan-2018	Structure and assumptions changed
Eliza Kowalczyk	17-Jan-2018	Upsampling documentation update
Sylwia Mól	19-Jan-2018	Logo added
Malwina Molendowska	22-Jan-2018	Reconstruction documentation update
Alicja Martinek	22-Jan-2018	mod5 documentation + lib
Karolina Gajewska	22-Jan-2018	Brain3D implementation description added
Mateusz Pabian	23-Jan-2018	DTI description added

2. Assumptions

The aim of this project is to create the system for MRI data pre-processing and post-processing using Python, SciPy, Cython and Qt5 (detailed description of tools available in chapter *Implementation*, section *Tools*). The project is divided into 11 modules (more: chapter *Structure*).

The application is called SieMRI (the abbreviation for Siemens MRI). It allows to recontruct MRI images, correct intensity inhomogenities, estimate non-stationary noise and remove it using one of two methonds, show diffusion tensors, strip the skull, show specific parts of human brain, upsample the image or visualize brain in 3D or using specific plane. MRI image reconstruction starts automatically when user choose the file with data. Other functionalities are non-obligatory - the user decide if specific functionality should be used.

3. Structure

As mentioned in chapter *Assumptions* the project is divided into 11 modules. Each module includes one system‘s functionality, as follows:

- Module 1.: MRI images reconstruction using Cartesian SENSE algorithm.
- Module 2.: Intensity inhomogenity correction on MRI reconstructed images.
- Module 3.: Non-stationary noise estimation (a homomorphic approach)
- Module 4.: Non-stationary noise filtering #1 with LMMSE filter.
- Module 5.: Non-stationary noise filtering #2 using unbiased non-local means filter.
- Module 6.: Diffusion tensor imaging (WLS, NLS algorithms)
- Module 8.: Skull stripping
- Module 9.: MRI segmentation of the human brain
- Module 10.: Upsampling based on non-local means approach
- Module 11.: Brain 3D visualization using marching cubes method
- Module 12.: Oblique imaging

Module 7. is excluded due to limited human resources and time.

The input and output signals for each module are listed in the table.

Module	Input	Output	Before that module
1	**k**-space signals	**x**-space fully reconstructed data	———
2	reconstruccted image	image with intensity correction	image reconstruction
3	reconstructed image	estimated noise map	reconstruction
4	reconstructed, normalized image	Rician noise-free image	intensity correction
5	image, noise map	Rician noise-free image	reconstruction, noise estimation
6	reconstructed, normalized image; gradient-sequence vectors	estimated diffusion tensor 3D 6-channel image	modules 1-5, 8*
8	reconstructed, normalized, filtered image	image without non-brain tissues	modules 1- 4 or 5
9	image without non-brain tissues	segmentated image	
10	320x240 image	640x480 image	denoised data
11	segmentated image	new window of application	segmentation
12	many images from one examination	one image from non-standard perspective	denoised data

Dependencies between modules are shown on the picture. 11th module is divided into two parts, because every part of module requires other pre-procesing activities.

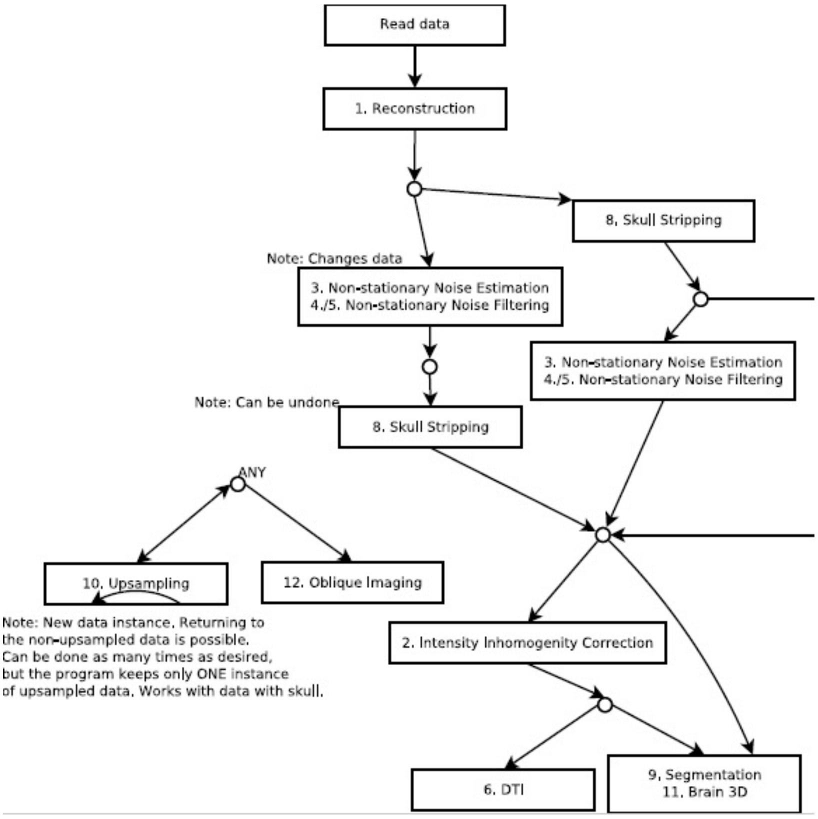
Read data
1. Reconstruction
8. Skull Stripping
Note: Changes data
3. Non-stationary Noise Estimation
4./5. Non-stationary Noise Filtering
3. Non-stationary Noise Estimation
4./5. Non-stationary Noise Filtering
Note: Can be undone
8. Skull Stripping
ANY
10. Upsampling
12. Oblique Imaging
Note: New data instance. Returning to
the non-upsampled data is possible.
Can be done as many times as desired,
but the program keeps only ONE instance
of upsampled data. Works with data with skull.
2. Intensity Inhomogenity Correction
6. DTI
9. Segmentation
11. Brain 3D

Figure 3.1. Dependencies between modules

4. User guide

4.1. Requirements

what user need to use this app - e.g. windows version etc
Currently conda is sufficient.

4.2. Instruction

instructions for user - GUI screens etc

5. Detailed description

5.1. Module 1. MRI reconstruction

The aim of this module is to formulate mathematical algorithm, which enables proper data reconstruction for images obtained with parallel MRI scans. The reconstruction is performed with use of Sensitivity Encoding (SENSE) algorithm in least squares (LS) solution context and Tikhonov regularization method.

Generally, parallel MRI acquisitions are targeted to diminish time needed for data sampling. The usage of multiple coils enabled simultaneous acquisition of signals. A further step, which is acquiring partial data from **k**-space, leads to craved time savings, meanwhile maintaining full spatial resolution as well as contrast at the same time. However, the approach of omitting lines in acquisition step results in data aliasing, i.e. folded images that need further data processing.

To clearly mark out how data is processed in this module, we list following reconstruction steps: i) the application of 2D Fourier Transform transform (2D FFT) to **k**-space data (acquired raw signals) from multiple coils. The result is a set of **x**-space images with folded pixels, ii) the sensitivity maps estimation of coils profiles (the information is needed to properly unfold subsampled data) and iii) the proper unfolding data process with usage of SENSE reconstruction algorithm and its alterations.

The most crucial step in processing is estimation of sensitivity coil profiles as a successful image reconstruction with use of pMRI algorithms highly depends on accurate sensitivity coil assessment. As sensitivity information varies from scan to scan it is impossible to obtain absolute maps. To obtain reliable knowledge, reference scans have to be conducted each time an examination is performed. These low-resolution information helps to estimate coil profiles with use of the many methods i.e. dividing each component coil image by a 'sum of squares' image.

It basic formulation, SENSE algorithm is applied to Cartesian MRI data undersampled uniformly by a factor r (i.e. $r = 2$ means that every other line in **k**-space is skipped). After Fourier transformation, each pixel in **x**-space image received in l-th coil can be seen as weighted sum of r pixels from full FOV, each multiplied by corresponding localized values of maps. The distance between those 'aliased' points in the full FOV is always equal to the desired FOVy value divided by subsampling rate. Obviously, depending on subsampling rate the number of folded pixels changes. Basically, the signal in one pixel at a certain location (x, y) received from l-th component coil image D_l^S with chosen subsampling rate r can be written as:

$$D_l^S(x,y) = S_l(x,y_1)D^R(x,y_1) + S_l(x,y_2)D^R(x,y_2) + ... + S_l(x,y_r)D^R(x,y_r), \tag{5.1}$$

where index l counts from 1 to L (number of coils) and index i counts from 1 to r. Eq.(5.1) can be rewritten as:

$$D_l^S(x,y) = \sum_{i=1}^{r} S_l(x,y_i)D^R(x,y_i) \quad \text{for} \quad l = 1, ..., L. \tag{5.2}$$

Including all L coils the above equation can be rewritten in a matrix form:

$$\mathbf{D}^S(\mathbf{x}) = \mathbf{S}(\mathbf{x})\mathbf{D}^R(\mathbf{x}), \tag{5.3}$$

The vector $\mathbf{D}^S(\mathbf{x})$ denotes the aliased coil image values at a specific location **x** = (x, y_i) and has a length of L, $\mathbf{S}(\mathbf{x})$ is a LxR matrix and represents the sensitivities values for each coil at the r superimposed positions and $\mathbf{D}^R(\mathbf{x})$ lists the r pixels from full FOV image to be reconstructed. The closed-form solution of the problem is as follows:

$$\widehat{\mathbf{D}^R(\mathbf{x})} = (\mathbf{S}^H(\mathbf{x})\mathbf{S}(\mathbf{x}))^{-1}\mathbf{S}^H(\mathbf{x})\mathbf{D}^S(\mathbf{x}), \tag{5.4}$$

where $\widehat{\mathbf{D}^R(\mathbf{x})} = [D^R\widehat{(x, y_1)}, ..., D^R\widehat{(x, y_r)}]^T$ and $\mathbf{S}^H(\mathbf{x})$ is the conjugate transpose of the $\mathbf{S}(\mathbf{x})$ matrix. The final reconstruction image is defined as:

$$M(\mathbf{x}) = \left|\widehat{\mathbf{D}^R(\mathbf{x})}\right|. \tag{5.5}$$

The ‘unfolding’ process can be performed as long as inversion of $\mathbf{S}(\mathbf{x})$ matrix is possible. Therefore, we cannot set the value of subsampling rate exceeding the number of coils L. To restore full FOV data, SENSE algorithm has to be recalled for each pixel in aliased **x**–space image.

A regularization approach is defined as an inversion method that introduces additional information in order to stabilize the solution. This method is beneficial as it roughly matches the desired solution and is less sensitive to perturbations of the data. Tikhonov regularization is a common approach to obtain an inexact solution to a linear system of equations. In particular, the Tikhonov regularized estimate reads as follows:

$$\widehat{\mathbf{D}^R_{reg}} = \operatorname*{argmin}_{\mathbf{D}^R} \left\{ \left\| \mathbf{D}^S - \mathbf{S}\mathbf{D}^R \right\|^2 + \lambda^2 \left\| \mathbf{A}(\mathbf{D}^R - \mathbf{D}) \right\|^2 \right\}, \tag{5.6}$$

where λ is a regularization parameter ($\lambda > 0$) and **D** is a prior image known as a regularization image. Selection of the parameter λ and **D** can be performed using different procedures. In this module **A** is assumed to be an identity matrix. The first term provides fidelity to the data and the second introduces prior knowledge (e.x. median filtered initial guess of LS SENSE) about the expected behaviour of $\mathbf{D}^R$. The Tikhonov regularization problem is given by:

$$\widehat{\mathbf{D}^R_{reg}} = \mathbf{D} + (\mathbf{S}^H\mathbf{S} + \lambda\mathbf{A}^H\mathbf{A})^{-1}\mathbf{S}^H(\mathbf{D}^S - \mathbf{S}\mathbf{D}). \tag{5.7}$$

A reasonable value for λ can be picked using many technique, i.e. the L-curve criterion or generalized cross-validation.

Module input: Synthetic MR images are brain MRI slices coming from BrainWeb are normalized to [0-255] (all with intensity non-uniformity INU=0). Only T1- and T2-weighted data is used. The dataset is free of noise and the background areas are set to zero. The slice thickness equals 1 mm. These images are used then to simulate synthetic noisy accelerated parallel Cartesian SENSE MRI data according to following steps (the data simulation is performed with use of eight receiver coils ($L = 8$)): i) simulated sensitivity maps (divided into the ratio 3:1 for real and imaginary parts, respectively) are added to fully-sampled **x**-space data, ii) correlated complex Gaussian noise with different values of standard deviations is added to each coil image, iii) 2D FFT and data subsampling with chosen reduction factor r is performed and iv) 2D iFFT is applied to recover data in **x**-space. Then, data reconstruction process is conducted.

Module output: The output is full resolution reconstructed data performed with two different algorithms: SENSE (LSE) and Tikhonov regularization.

5.2. Module 2. Intensity inhomogeneity correction

The Intensity inhomogeneity of the same tissue varies with the location of the tissue within the image. In other words it refers to the slow, nonanatomic intensity variations of the same tissue over the image domain. It can be due to imaging instrumentation (such as radio-frequency nonuniformity, static field inhomogeneity, etc.) or the patient movement. This artifact is particularly severe in MR images captured by surface coils. Although intensity inhomogeneity is usually hardly noticeable to a human observer, many medical image analysis methods, such as segmentation and registration, are highly sensitive to the spurious variations of image intensities.

The aim of this module is estimation of intensity inhomogeneity field in MR image using surface fitting method. The method fits a parametric surface to a set of image features that contain information on intensity inhomogeneity. The resulting surface, which is usually polynomial or spline based, represents the multiplicative inhomogeneity field that is used to correct the input image.

The steps of this approach are:

1. Extract a background image from the corrupted MRI image, for example, by smoothing the image with a Gaussian filter of a large bandwidth (about 2/3 the size of the MRI image) to filter out all the image details that correspond to highfrequency components.

2. Select few data points from the background image and save their coordinates and graylevel values into a matrix $D = (xi, yi, gi), i = 1, 2, ...n$. It is recommended not to select points from the regions where there is no MRI signal since this regions has no bias field signal.

3. Select a parametric equation for the fitted surface . It is better to fit simple surfaces such as low order polynomial surfaces since they are very smooth and their parameters are very easy to estimate.

4. Estimate the parameters of the surface that best fits the data in matrix D by the method of nonlinear least-squares.

5. Use the fitted equation to generate an image of the bias field signal.

6. Divide the corrupted MRI image by the estimated bias field image in step 5.

Even though different surfaces can reasonably fit the data very well and it is not possible to tell which surface is most likely represents the actual bias field signal, however, in practice the bias signal estimated by fitting a smooth 2-dimensional polynomial surface to a background image can be used effectively to restore the corrupted MRI image.

Fitting of the surface can be done by means of the Levenberg-Marquardt algorithm for nonlinear least squares fitting of a function $f(x, y; a_1, ..., a_m)$ of known form to n data points $(x_1, y_1, g_1), ..., (xn, yn, gn)$. For example, a polynomial surface of degree three can be fitted which has the following equation: $f(x, y; a) = a_1x^3 + a_2y^3 + a_3$, where $a = a_1, a_2, a_3$ is the parameter vector that define the surface. If we substitute the data points in the nonlinear function we get an overdetermined set of equations, i.e.,

$$\left\{\begin{matrix} g_1 = f(x_1, y_1; a_1, a_2, ..., a_m) \\ . \\ . \\ g_n = f(x_n, y_n; a_1, a_2, ..., a_m) \end{matrix}\right\} \tag{5.8}$$

These equations can be solved to obtain the unknown parameter vector $(a_1, a_2, ..., a_m)$ by minimizing the sum of the squares of the differences between the data and the fitted function

$$Q(\mathbf{a}) = \frac{1}{2}\sum_{i=1}^{n}(g_i - f(x_i, y_i; a_1, a_2, ..., a_m))^2 \tag{5.9}$$

Let $r_i(\mathbf{a}) = (g_i - f(x_i, y_i; a_1, a_2, ..., a_m)$, which is the residual vector of point i, then equation(??) can be written as:

$$Q(\mathbf{a}) = \frac{1}{2}\sum_{i=1}^{n}(r_1(\mathbf{a}))^2 \tag{5.10}$$

According to the Levenberg-Marquardt algorithm, Eq(??) can be solved iteratively to find the values of the parameters vector (**a**) starting from an initial estimate of the parameter vector $(\mathbf{a}_0)$ using:

$$\mathbf{a}_{i+1} = \mathbf{a}_i - (H + \lambda diag[H])^{-1} \bigtriangledown Q(\mathbf{a}_i) \tag{5.11}$$

where H and $\bigtriangledown$Q($\mathbf{a}_1$) are Hessian matrix and the gradient of Eq. ?? both evaluated at ai, diag is the diagonal elements of the Hessian matrix. At each iteration, the algorithm tests the value of the residual error and adjusts λ accordingly.

List of References

[1], [2]

5.3. Module 3. Non-stationary noise estimation

Magnetic Resonance Imaging (MRI) is known to be affected by several sources of quality deterioration, due to limitations in the hardware, scanning times, movement of patients, or even the motion of molecules in the scanning subject. Among them, noise is one source of degradation that affects acquisitions. The presence of noise over the acquired MR signal is a problem that affects not only the visual quality of the images, but also may interfere with further processing techniques such as registration or tensor estimation in Diffusion Tensor MRI.

The aim of this module is estimation of non-stationary noise maps based on MRI image. The module contains a homomorphic method for the non-stationary noise of Gauss and Rice aswell. For the high SNR (signal to noise ratio) algoritm for Gaussian case should be used and for the lower value algorithm for Rician case. In the first case image is corrupted with Gaussian noise which has mean equal to zero spatially-dependent variance $\sigma^2(x)$:

$$I(x) = A(x) + N(x; 0, \sigma^2(x)) = A(x) + \sigma(x) \cdot N(x; 0, 1) \tag{5.12}$$

The goal is to estimate $\sigma(x)$ from the final image $I(x)$.

The variance of the noise $\sigma^2(x)$ slowly differs across the image. Therefore the first step is to remove mean of the image in order to avoid contribution of $A(x)$:

$$I_n(x) = I(x) - E\{I(x)\} = \sigma(x) \cdot N(x; 0, 1) \tag{5.13}$$

where $E\{I(x)\}$ is expected value in each point in the image. It is estimated as local mean using 3×3 window.

The next step is to separate signals $\sigma(x)$ and $N(x)$ by applying the logarithm, but to do so the absolute value of $I_n(x)$ is needed:

$$log\,|I_n(x)| = log\sigma(x) + log\,|N(x)| \tag{5.14}$$

where $log\left|N(x)\right|$ has its energy distributed in all frequencies and $log\sigma(x)$ is a low frequeny signal.

$log\sigma(x)$ is the interesting part, so to get rid of $log\left|N(x)\right|$ low pas filter has to be used:

$$LPF\left\{log\left|I_n(x)\right|\right\} \approx log\sigma(x) + \delta_N \tag{5.15}$$

where δ_N is a lowa pass resideu of $log\left|N(x)\right|$ that must be removed from the estimation. Throughout various calculations the final outcome of the LPF was delineated as:

$$LPF\left\{log\left|I_n(x)\right|\right\} \approx log\sigma(x) - log\sqrt{2} - \frac{\gamma}{2} \tag{5.16}$$

with γ being Euler-Mascheroni constant.

Taking the exponential of every part of foregoing formula estimator of $\sigma(x)$ can be defined as:

$$\widehat{\sigma(x)} = \sqrt{2}e^{LPF\{log|I(x)-E\{I(x)\}|\}+\gamma/2} \tag{5.17}$$

The Rician case is similar to the described abocv Gaussian case, but a bit more complicated. The signal $A(x)$ is corrupted with complex Gaussian noise with zero mean and spatially-dependent variance $\sigma^2(x)$, which has module following a nonstationary Rician distribution:

$$I(x; A(x), \sigma(x)) = \left|A(x) + N_r(x; 0, \sigma^2(x)) + j \cdot N_i(x; 0, \sigma^2(x))\right| \tag{5.18}$$

To simplify, the dependence of $I(x)$ with $A(x)$ and $\sigma(x)$ are removed.

First couple of steps in Rician case are congenial to the steps taken in Gaussian case. The local mean calulated using 3×3 window is substracted from the image so it can be centered. After that logarithm of the absolute value of centered image is filtered by low passing filter, giving as a result:

$$log\left|I_n(x)\right| = log\sigma(x) + log\left|G(s_0(x))\right| LPF\left\{log\left|I_n(x)\right|\right\} \approx log\sigma(x) + \delta_R \tag{5.19}$$

where σ_R is a low pass residue of $log\left|G(s_0(x))\right|$. Again after various operation previouse formula presents as:

$$LPF\left\{log\left|I_n(x)\right|\right\} \approx log\sigma(x) - log\sqrt{2} - \frac{\gamma}{2} + \varphi(s_0(x)) \tag{5.20}$$

with $\varphi(s_0(x))$ being a Rician/Gaussian correction function. Finally the estimator for $\sigma(x)$ is defined as:

$$\widehat{\sigma(x)} = \sqrt{2}e^{LPF\{log|I(x)-E\{I(x)\}|\}}e^{\gamma/2-\varphi(s_0(x))} \tag{5.21}$$

5.4. Module 4. Non-stationary noise filtering 1

The aim of this module is to remove Rician noise from MR images. If both real and imaginary parts of signal are corrupted with zero-mean uncorrelated Gaussian noise with equal variance, the envelope of magnitude signal will follow a Rician distribution. Many processes allow to remove noise, here the method is used to denoise MR images is the linear minimum square error estimator (LMMSE). The main purpose of LMMSE is to find a closed-form estimator for a signal that follows a Rician distributions. It is more efficient than optimization-based solutions. The estimator uses information of the sample distribution of local statistics of the image such as the local mean, the local variance and the local mean square value. In this method, the true value for each noisy pixel is estimated by a set of pixels selected from a local neighborhood.

The LMMSE estimator for a 2-D signal with Rician distribution is defined:

$$\widehat{A_{ij}^2} = E\{A_{ij}^2\} + C_{A_{ij}^2 M_{ij}^2} C^{-1}_{M_{ij}^2 M_{ij}^2} (M_{ij}^2 - E\{M_{ij}^2\}) \tag{5.22}$$

where A_{ij} is the unknown intensity value in pixel ij, M_ij the observation vector, $C_{A_{ij}^2 M_{ij}^2}$ the cross-covarience vector and $C_{M_{ij}^2 M_{ij}^2}$ the covarience matrix. If the estimator is simplified to be pointwise, vectors and matrics become scalar values. Then the assuming local ergodicity with a square nighbourhood around the pixel ij, the finally equation for LMMSE is defined:

$$\widehat{A_{ij}^2} = \langle M_i j^2 \rangle - 2\sigma_n^2 + K_{ij}(M_{ij}^2 - \langle M_{ij}^2 \rangle) \tag{5.23}$$

with K_{ij}

$$K_{ij}^2 = 1 - \frac{4\sigma_n^2(\langle M_i j^2 \rangle - 2\sigma_n^2)}{\langle M_i j^4 \rangle - \langle M_i j^2 \rangle^2} \tag{5.24}$$

The LMMSE estimator is related to the quality of the estimate of the noise variance σ_n^2. For the noise estimation the mode of the sample mean is used:

$$\widehat{\sigma_n} = \sqrt{\frac{2}{\pi}} mode(\widehat{\mu_1}_{ij}) \tag{5.25}$$

where $\widehat{\mu_1}_{ij}$

$$\widehat{\mu_1}_{ij} = \frac{1}{|\eta_{ij}} \sum_{p:\ \eta_{ij}} I_p \tag{5.26}$$

The use of the LMMSE method should makes the filtering process computationally far more efficient and easier to implement. Also the use of local statistics should decrease estimator dependent of parameters such as the size of window.

Module input: Reconstructed, normalized and corrected data.

Module output: Image without Rician noise.

5.5. Module 5. Non-stationary noise filtering 2

Magnetic Resonance images are endangered of being corrupted by noise and artifacts. Since they are used as a basis for medical diagnosis their quality has to be at highest possible level. Noise can be dealt with by changing the parameters of images acquisition, however it increases the scanning time, which is undesirable in medical imaging. To overcome this obstacle, post-processing methods like filtering are employed for denoising. In domain of MRI denoising many filters may be used, though here emphasis is put on Unbiased Non-Local Means (UNLM) filter, which is an extension of NLM filter. In order to understand Unbiased version of this algorithm, the basic one has to be presented.

Having image *Y*, the NLM algorithm calculates the new value of point *p* accordingly to the equation:

$$\begin{gathered} NLM(Y(p)) = \sum_{\forall q \in Y} w(p,q)Y(q) \\ 0 \leq w(p,q) \leq 1, \sum_{\forall q \in Y} w(p,q) = 1 \end{gathered} \tag{5.27}$$

It can be seen that value of *p* is calculated as weighted average of pixels in the image (*q*), having fulfilled restrictions from 5.27. To determine before mentioned average the similarity between square neighbourhoods widows centered around pixels *p* and *q* are calculated. The size of the window can determined by the user, defined by parameter R_{sim}. Equation 5.28 shows how to determine this similarity.

$$w(p,q) = \frac{1}{Z(p)} e^{\frac{d(p,q)}{h^2}} \tag{5.28}$$

Z(p) is the normalizing constant which also uses exponential decay parameter h and the weighted Euclidean distance measure for pixels in each neighbourhood, called *d*.

$$Z(p) = \sum_{\forall q} e^{\frac{d(p,q)}{h^2}} \tag{5.29}$$

$$d(p,q) = G_p ||Y(N_p) - Y(N_q)||^2_{R_{sim}} \tag{5.30}$$

In above equation G_p stands for a Gaussian weighting function that has a 0 mean and standard deviation usually equal to 1.

Once NLM filter is fully explained, unbiased extension of it can be examined. It builds on the properties of MRI signal. According to [3] the magnitude signal of MRI follows a Rician distribution. Furthermore, for low intensity regions the Rician distribution approaches to a Rayleigh one, whilst for high intensity it shifts towards Gaussian. It was investigated that this bias can be handled by filtering the squared MRI image, since it is not longer signal-dependent [3]. As a consequence the bias, which equals $2\sigma^2$ [4] can be deleted with ease. The blueprint for UNLM can be summarized in:

- noise estimation - which can be done by calculating standard deviation of background in the image. To distinguish background and the body on the MRI scan the Otsu thresholding method [5] can be successfully used. In this version noise maps are used, to achieve non-stationary noise filtration,
- calculating NLM values for each point of image as in 5.27,
- assessing the unbiased value of each point accordingly to the equation 5.31.

$$UNLM(Y) = \sqrt{NLM(Y)^2 - 2\sigma^2} \tag{5.31}$$

In above equation σ refers to value of non-stationary noise in the signal, more precisely it is a value for each pixel from the original image, stored in a form of noise map.

UNML implementation for diffusion weighted data becomes a bit less trivial task, due to the new dimension of data associated with different gradients for each slice. Based on assumption that gradients in similar directions present related behaviours, UNLM for DWI can be formulated as:

$$Y_i(p) = \sqrt{\sum_{j \in \Theta_i^N} \sum_{q \in N_p} w_i^j(p,q) M_j^2(q) - 2\sigma^2} \tag{5.32}$$

Where weights are calculated as for structural data and *M* is a vector containing gray values.

$$d_i^j(p,q) = (M_i(N_p) - M_j(N_q))^T G_p (M_i(N_p) - M_j(N_q)) \tag{5.33}$$

However it was reported in [6] that denoising diffusion weighted data using UNLM method gives no significant results, so gradients related data can be ignored, or filtered as structural data.

It is worth mentioning that UNLM filter's performance is highly dependent on parameter values. The optimal values of them were examined in [3] and same values are adapted in presented implementation. Properly-tuned filter can significantly increase SNR of the scans while preserving body structures.

Module input: Previously reconstructed, normalized and corrected data, noise maps.

Module output: Image with deleted Rician noise by unbiased non-local means filter.

5.6. Module 6. Diffusion tensor imaging

The aim of this module is to estimate brain tissue diffusion tensor from Diffusion Weighted Images (DWI). DWI are obtained using a different pulse sequence than anatomical MRI images and as such differ in their information content. Concretely, Diffusion Tensor Imaging (DTI) gives an insight into tissue microstructure based on DWI, which are obtained by probing tissues using gradient-pulse excited water molecules [7]. This enables indirect measurements of structural orientation and the degree of anisotropy as water molecules diffuse differs between tissues.

In DTI-MRI the measured signal is defined as [8]:

$$S\left(b, \boldsymbol{g}\right) = S_0 exp\left(-b\boldsymbol{g^T}\boldsymbol{D}\boldsymbol{g}\right) \tag{5.34}$$

where $S\left(b, \boldsymbol{g}\right)$ is the measured signal, S_0 is the reference signal without diffusion gradient attenuation, b is the diffusion weight scalar, $\boldsymbol{g}$ is the diffusion encoding gradient vector of unit length and $\boldsymbol{D}$ is the diffusion tensor.

The diffusion tensor $\boldsymbol{D}$ is symmetric and describes molecular mobility along each direction:

$$\boldsymbol{D} = \begin{bmatrix} D_{xx} & D_{xy} & D_{xz} \\ D_{yx} & D_{yy} & D_{yz} \\ D_{zx} & D_{zy} & D_{zz} \end{bmatrix} \tag{5.35}$$

Furthermore, the Eq. (5.34) can be rewritten as:

$$lnS\left(b, \boldsymbol{g}\right) = lnS_0 - b\boldsymbol{g^T}\boldsymbol{D}\boldsymbol{g} \tag{5.36}$$

which established a linear relationship between model parameters and measured quantity.

Estimating $\boldsymbol{D}$ from the above equation can be done using Weighted Least Squares (WLS) or Nonlinear Least Squares (NLS) algorithms. In order to correctly estimate the diffusion tensor it is necessary to acquire at least seven DWI - one for each direction of diffusion and one in the absence of diffusion gradient.

Let γ be the parameters vector of DTI model:

$$\boldsymbol{\gamma} = \left[lnS_0, D_{xx}, D_{yy}, D_{zz}, D_{xy}, D_{yx}, D_{xz}\right]^T \tag{5.37}$$

Furthermore let $\boldsymbol{W_i}$ be the design matrix of our model. In this case it represents the diffusion vector in a given direction for i-th measurement:

$$\boldsymbol{W_i} = \left[1, -b_i {g_{ix}}^2, -b_i {g_{iy}}^2, -b_i {g_{iz}}^2, -2b_i g_{ix} g_{iy}, -2b_i g_{iy} g_{iz}, -2b_i g_{ix} g_{iz}\right] \tag{5.38}$$

Then equations 5.34 oraz 5.36 can be rewritten as:

$$S\left(\gamma\right) = exp\left(\boldsymbol{W_i}\gamma\right) \tag{5.39}$$

$$ln\left(S\left(\gamma\right)\right) = \boldsymbol{W_i}\gamma \tag{5.40}$$

The aforementioned expressions can be used to estimate the vector $\boldsymbol{\gamma}$ using Least Squares Regression models. It is important to note that tensor built using the elements of $\boldsymbol{\gamma}$ will be symmetric.

5.6.1. Weighted Least Squares estimation

For Weighted Least Squares (WLS) model the cost function is defined as [9]:

$$f_{WLS}\left(\boldsymbol{\gamma}\right) = \frac{1}{2}\sum_{i=1}^{m}{\omega_i}^2\left(y_i - \sum_{j=1}^{7}W_{ij}\gamma_j\right)^2 \quad ; \quad y_i = ln(S(\gamma)_i) \tag{5.41}$$

where m is the total number of measurements while ω is the measurement weight associated with WLS method.

The aim of Least Squares Regression methods is to minimaze the cost function f_{WLS} with respect to $\boldsymbol{\gamma}$, which coincides with the function gradient $|\nabla f_{WLS}|$ equal to zero.

In general, vector-valued function gradient is defined as a vector whose elements can be calculated using the following formula:

$$|\nabla f_{WLS}|_k = \frac{\partial f_{WLS}}{\partial \gamma_k} \tag{5.42}$$

For WLS cost function:

$$\begin{aligned}
\frac{\partial f_{WLS}}{\partial \gamma_k} &= \sum_{i=1}^{m}\frac{\partial}{\partial \gamma_k}{\omega_i}^2\left(y_i - \sum_{j=1}^{7}W_{ij}\gamma_j\right) \\
&= \sum_{i=1}^{m}{\omega_i}^2\left(y_i - \sum_{j=1}^{7}W_{ij}\gamma_j\right)\frac{\partial}{\partial \gamma_k}\left(y_i - \sum_{j=1}^{7}W_{ij}\gamma_j\right) \\
&= \sum_{i=1}^{m}{\omega_i}^2\left(y_i - \sum_{j=1}^{7}W_{ij}\gamma_j\right)\sum_{j=1}^{7}\left(-W_{ij}\right)\frac{\partial}{\partial \gamma_k}\gamma_j \\
&= \sum_{i=1}^{m}{\omega_i}^2\left(y_i - \sum_{j=1}^{7}W_{ij}\gamma_j\right)\left(-W_{ik}\right)
\end{aligned} \tag{5.43}$$

or in matrix form:

$$\nabla f_{WLS} = -\boldsymbol{W}^T\boldsymbol{\omega}^T\boldsymbol{\omega}\left(\boldsymbol{y} - \boldsymbol{W}\boldsymbol{\gamma}\right) \tag{5.44}$$

where $\boldsymbol{\omega}$ is a diagonal matrix with non-zero elements equal to WLS model weights.

The Hessian of vector-valued function is defined as a matrix whose elements computed as:

$$\left|\nabla^2 f_{WLS}\right|_{lk} = \frac{\partial^2 f_{WLS}}{\partial \gamma_l \partial \gamma_k} \tag{5.45}$$

For WLS cost function:

$$\begin{aligned}
\frac{\partial^2 f_{WLS}}{\partial \gamma_l \partial \gamma_k} &= \frac{\partial}{\partial \gamma_l}\frac{\partial f_{WLS}}{\partial \gamma_k} \\
&= \frac{\partial}{\partial \gamma_l}\sum_{i=1}^{m}{\omega_i}^2\left(y_i - \sum_{j=1}^{7}W_{ij}\gamma_j\right)\left(-W_{ik}\right) \\
&= \sum_{i=1}^{m}{\omega_i}^2\frac{\partial}{\partial \gamma_l}\left(y_i - \sum_{j=1}^{7}W_{ij}\gamma_j\right)\left(-W_{ik}\right) \\
&= \sum_{i=1}^{m}{\omega_i}^2\left(-W_{il}\right)\left(-W_{ik}\right) \\
&= \sum_{i=1}^{m}{\omega_i}^2 W_{il}W_{ik}
\end{aligned} \tag{5.46}$$

or in matrix form:

$$\nabla^2 f_{WLS} = \boldsymbol{W}^T \boldsymbol{\omega}^T \boldsymbol{\omega} \boldsymbol{W} \tag{5.47}$$

One can notice that Eq. (5.44) does not depend on model parameters, meaning that it can be solved using it's normal equation:

$$\begin{aligned} \nabla f_{WLS} = 0 \iff -\boldsymbol{W}^T \boldsymbol{\omega}^T \boldsymbol{\omega} \left(\boldsymbol{y} - \boldsymbol{W}\boldsymbol{\gamma}\right) &= 0 \\ \boldsymbol{W}^T \boldsymbol{\omega}^T \boldsymbol{\omega} \boldsymbol{W} \boldsymbol{\gamma} &= \boldsymbol{W}^T \boldsymbol{\omega}^T \boldsymbol{\omega} \boldsymbol{y} \\ \boldsymbol{\gamma} = \left(\boldsymbol{W}^T \boldsymbol{\omega}^T \boldsymbol{\omega} \boldsymbol{W}\right)^{-1} & \boldsymbol{W}^T \boldsymbol{\omega}^T \boldsymbol{\omega} \boldsymbol{y} \end{aligned} \tag{5.48}$$

Before continuing, it is important to note that there exists no simple choice of $\boldsymbol{\omega}$ weights. The authors of [9] suggest using the measured signal $\boldsymbol{S}$, in [10] one can read that weights should be the Linear Least Squares estimate of signal in each voxel, while the authors [11] argue for the use of signal variance estimation using Rice's noise estimate. As one can see, WLS model complexity varies depending on the choice of weights vector computation method.

5.6.2. Nonlinear Least Squares estimation

Nonlinear Least Squares cost function is defined as:

$$f_{NLS}\left(\gamma\right) = \frac{1}{2}\sum_{i=1}^{m}\left(S_i - exp\left(\sum_{j=1}^{7} W_{ij}\gamma_j\right)\right)^2 = \frac{1}{2}\sum_{i=1}^{m} r_i^2 \tag{5.49}$$

Same as before, we will derive NLS cost function gradient and Hessian matrix.

$$\begin{aligned} \frac{\partial f_{NLS}}{\partial \gamma_k} &= 2\frac{1}{2}\sum_{i=1}^{m} r_i \frac{\partial}{\partial \gamma_k}\left(r_i\left(\gamma\right)\right) \\ &= \sum_{i=1}^{m}(-r_i)\frac{\partial}{\partial \gamma_k}\left(exp\left[\sum_{j=1}^{7} W_{ij}\gamma_j\right]\right) \\ &= \sum_{i=1}^{m}(-r_i)exp\left(\sum_{j=1}^{7} W_{ij}\gamma_j\right)\frac{\partial}{\partial \gamma_k}\left(\sum_{j=1}^{7} W_{ij}\gamma_j\right) \\ &= \sum_{i=1}^{m}(-r_i)exp\left(\sum_{j=1}^{7} W_{ij}\gamma_j\right)\left(\sum_{j=1}^{7} W_{ij}\frac{\partial}{\partial \gamma_k}\gamma_j\right) \\ &= \sum_{i=1}^{m}(-r_i)exp\left(\sum_{j=1}^{7} W_{ij}\gamma_j\right) W_{ik} \\ &= \sum_{i=1}^{m}(-r_i)\hat{S}_i W_{ik} \end{aligned} \tag{5.50}$$

where $\hat{S}_i$ is the voxel signal estimate of NLS model. Let $\hat{\boldsymbol{S}}$ be a diagonal matrix with non-zero elements set to $\hat{S}_i$. Then f_{NLS} computation can be rewritten in matrix form:

$$\nabla f_{NLS} = -\boldsymbol{W}^T \hat{\boldsymbol{S}} \boldsymbol{r} \tag{5.51}$$

as one can see from the Eq. (5.51), the value of gradient depends on NLS signal estimate, meaning that contrary to WLS method, it is not possible to compute NLS estimate without resorting to iterative methods.

NLS Hessian matrix elements can be computed as:

$$\begin{aligned}
\frac{\partial^2 f_{NLS}}{\partial \gamma_l \partial \gamma_k} &= \frac{\partial}{\partial \gamma_l} \frac{\partial f_{NLS}}{\partial \gamma_k} \\
&= \frac{\partial}{\partial \gamma_l} \sum_{i=1}^{m} (-r_i) \hat{S}_i W_{ik} \\
&= -\sum_{i=1}^{m} \left(\hat{S}_i W_{ik} \frac{\partial}{\partial \gamma_l} r_i + r_i W_{ik} \frac{\partial}{\partial \gamma_l} \hat{S}_i \right) \\
&= -\sum_{i=1}^{m} \left(\hat{S}_i W_{ik} \frac{\partial}{\partial \gamma_l} \left[S_i - \hat{S}_i \right] + r_i W_{ik} \frac{\partial}{\partial \gamma_l} \hat{S}_i \right) \\
&= -\sum_{i=1}^{m} \left(\hat{S}_i W_{ik} \frac{\partial}{\partial \gamma_l} \left[-\hat{S}_i \right] + r_i W_{ik} \frac{\partial}{\partial \gamma_l} \hat{S}_i \right) \\
&= -\sum_{i=1}^{m} \left(\hat{S}_i W_{ik} \left[-\hat{S}_i W_{il} \right] + r_i W_{ik} \hat{S}_i W_{il} \right) \\
&= \sum_{i=1}^{m} \left(\hat{S}_i W_{ik} \hat{S}_i W_{il} - r_i W_{ik} \hat{S}_i W_{il} \right) \\
&= \sum_{i=1}^{m} W_{ki} \left(\hat{S}_i^{\,2} - r_i \hat{S}_i \right) W_{il}
\end{aligned} \tag{5.52}$$

or in matrix form:

$$\nabla^2 f_{NLS} = \boldsymbol{W}^T \left(\hat{\boldsymbol{S}}^{\boldsymbol{T}} \hat{\boldsymbol{S}} - \boldsymbol{R} \hat{\boldsymbol{S}} \right) \boldsymbol{W} \tag{5.53}$$

where $\boldsymbol{R}$ is the diagonal matrix whose non-zero elements are equal to NLS model residuals (r_i).

Usually, iterative approaches to function minimization compute the Taylor series of a given function and try to find a vector $\boldsymbol{\delta}$ such that $f_{NLS}(\boldsymbol{\gamma} + \boldsymbol{\delta}) < f_{NLS}(\boldsymbol{\gamma})$. If additionally:

$$\boldsymbol{\delta} = -\left(\nabla^2 f_{NLS} \right)^{-1} \nabla f_{NLS} \tag{5.54}$$

then this approach is called Newton's method in optimization.

5.6.3. Diffusion Biomarkers

Regardless of the chosen estimation method, the estimated vector $\hat{\boldsymbol{\gamma}}$ is reshaped to be a symmetric 3x3 matrix $\boldsymbol{D}$.

There exists a couple of strategies for visualizing this high-dimmensional array containing estimated tensor in each voxel of each slice. One of these approaches is to compute the eigenvalue decomposition of $\boldsymbol{D}$ and compute a series of images called Diffusion Biomarkers which can be used to extract additional information about tissue microstructure.

Let us define four biomarker images:

1. MD (Mean Diffusivity) is the mean of tensor eigenvalues. MD is an inverse measure of membrane density and is very similar for both gray (GM) and white matter (WM) and is higher for corticospinal fluid (CSF). MD is sensitive to cellularity, edema and necrosis [12].

$$MD = \frac{\lambda_1 + \lambda_2 + \lambda_3}{3} \tag{5.55}$$

2. RA (Relative Anisotropy) exhibits high degree of contrast between anisotropic (WM) and isotropic tissues.

$$RA = \sqrt{\frac{(\lambda_1 - MD)^2 + (\lambda_2 - MD)^2 + (\lambda_3 - MD)^2}{3\,MD}} \tag{5.56}$$

3. FA (Fractional Anisotropy) is a summary measure of microstructural integrity. It is highly sensitive to microstructural changes without considering the type of change [7].

$$FA = \sqrt{\frac{3}{2}}\sqrt{\frac{(\lambda_1 - MD)^2 + (\lambda_2 - MD)^2 + (\lambda_3 - MD)^2}{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}} \tag{5.57}$$

4. VR (Volume Ratio)

$$VR = \frac{\lambda_1 \lambda_2 \lambda_3}{MD^3} \tag{5.58}$$

where λ_1, λ_2, λ_3 are the tensor eigenvalues. One can notice that for each biomarker image to be real-valued, eigenvalues must be non-negative. There are a couple of strategies aimed at solving this problem [8]:

- Substituting negative eigenvalues with zero.
- Substituting eigenvalues with their absolute value.
- Computing Cholesky-parametrization of input vectors to WLS and NLS estimation methods.

One can also compute a 3D map using the eigenvectors corresponding to tensor eigenvalue of highest magnitude. The resulting map is a measure of the direction of principal diffusion, which along with it's magnitude (eigenvalue), is the input signal to Tractography.

Additionally, the aforementioned 3D map can be displayed as a RGB image, because eigenvectors obtained from eigenvalue decomposition of a symmetric real matrix are orthogonal. The intensity of resulting color image is then weighted using one of computed biomarker images, usually Fractional Anisotropy.

5.7. Module 8. Skull stripping

The aim of this module is to remove pieces of skull from MRI Image using at pleasure chosen algorithm from the literature. Skull stripping is performed with use of hybrid approach that combines watershed algorithms and deformable surface models.

Preliminary processing to isolate the brain form extra-cranial or non-brain tissues such as e.g. the eye sockets, skin from MRI head scans is commonly referred as skull stripping. Skull stripping methods which are available in the literature are broadly classified into five categories: mathematical morphology-based metods, intensity-based methods, deformable surface-based methods, atlas-based methods, and hybrid methods. Each skull stripping method has their own merits and limitations.

In this module is proposed a hybrid approach to robustly and automatically segment brain from non-brain tissues in T1-weighted MR image. The skull stripping consists of series of sequential steps:

1. Some relevant parameters are estimated from the input image (the coordinates of the brain COG, the average brain radius BR, an upper bound for the intensity of the cerebrospinal fluid CFS – white matter parameters).
2. Watershed algorithm is performed on the intensity image, with global minimum initialized within the cerebral white matter.

3. Deformable surface procedure to recover parts of cortex that may have been erroneously removed in watershed algorithm, using smoothness constraints on the shape of the skull and atlas information.

Watershed algorithms are based on image intensities. Typically, they attempt to locate the local maxima/minima of the norm of the image intensity gradient to segment the image into different connected components. The algorithm proceeds in two steps:

- Watershed transform - the sorting all of voxels (of the gray level inverted image) according to their intensity.

- Post-watershed correction - assessment the validity of the watershed segmentation and retrospectively correct it, because the result image is often inaccurate and nonsmooth, with extracerebral tissues and CSF frequently remaining.

Deformable surface algorithm used the watershed segmentation outputs a segmented volume with most of non-brain tissues removed. A deformable balloon-like template employes this brain volume. An initial template deformation is first completed using global parameters regarding the brain/non-brain border to roughly match the boundary of the brain. Next, the correctness of resulting surface is verified by an atlas-based analysis, and if important structures have been removed, it is modified.

5.8. Module 9. Segmentation

Brain MRI segmentation is an essential task in many clinical applications because it influences the entire analysis. This is because different processing steps rely on accurate segmentation of anatomical regions. For example, MRI segmentation is commonly used for measuring and visualizing different brain structures, for delineating lesions, for analysing brain development, and for image-guided interventions and surgical planning.

MRI Processing To prepare brain MRI for segmentation, it is necessary to perform several preprocessing steps. (Fig. 5.1).

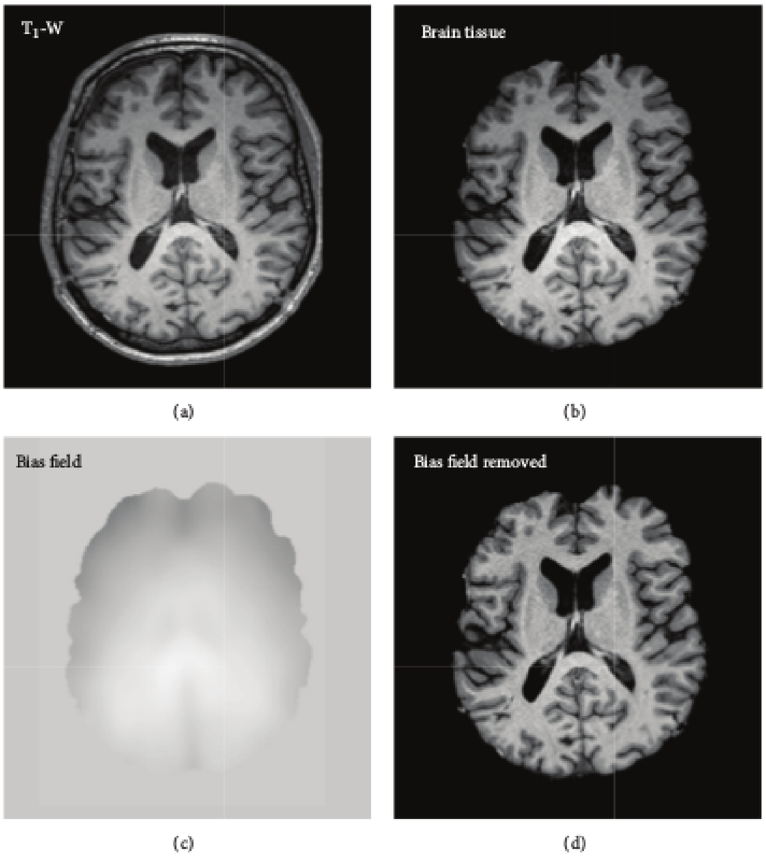

Figure 5.1. Triangulation for the 15 patterns.

The most important steps are: MRI bias field correction, image registration and brain extraction.

Basic An image for segmentation can be defined in 2D space (pixels) or in 3D space (voxels). Each image element is specified by its intensity value and coordinates for pixels (i,j) and for voxels (i,j,k). Intensity values are typically represented by a gray value 0, . . . , 255.

The goal of image segmentation is to divide an image into a set of semantically meaningful, homogeneous, and nonoverlapping regions of similar attributes such as intensity, depth, color, or texture. Thesegmentation result is either an image of labels identifying each homogeneous region or a set of contours which describe the region boundaries.

Fundamental components of structural brain MRI analysis include the classification of MRI data into specific tissue types and the identification and description of specific anatomical structures. The problems of segmentation and classification are interlinked because segmentation implies a classification, while a classifier implicitly segments an image. In the case of brain MRI, image elements are typically classified into three main tissue types: white matter (WM), gray matter (GM), and cerebrospinal fluid (CSF) (Fig. **??**).

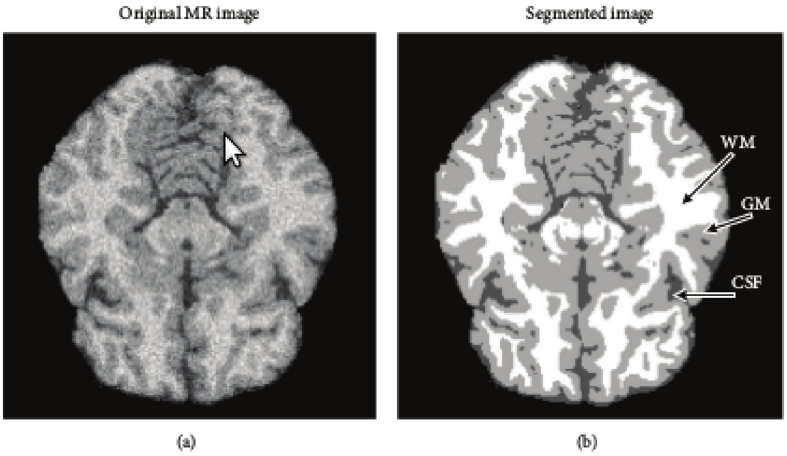

Figure 5.2. Triangulation for the 15 patterns.

One of the most important features for brain MRI segmentation is the intensity of brain tissue. However, intensity-based segmentation algorithms will lead to wrong results when intensity value are corrupted by MRI artifacts.

MRI Segmentation Methods There is no single method that can be suitable for all images, nor are all methods equally good for a particular type of image. For example, some of the methods use only the gray level histogram, while some integrate spatial image information to be robust for noisy environments. Some methods use probabilistic or fuzzy set theoretic approaches, while some additionally integrate prior knowledge (specific image formation model, e.g., MRI brain atlas) to further improve segmentation performance. The segmentation methods, with application to brain MRI, may be grouped as follows:

- manual segmentation;
- intensity-based methods (including thresholding, region growing, classification, and clustering);
- atlas-based methods;
- surface-based methods (including active contours and surfaces, and multiphase active contours);
- hybrid segmentation methods.

List of References

[13]

5.9. Module 10. Upsampling

Image spatial resolution is limited by several factors. There are for example time of acquisition, organ motions, signal to noise ratio or medical equipment which do not meet the specialized requirements. Specialists usually make relatively small number of 2-D slices. It takes less time but it also results in large slice thickness and spacing between slices. Another example are voxel-wise analysis. The method requires all image volumes of one subject to be analysed in one space. If they are not acquired from the same location (because of the organ motions) the interpolation must be involved.

Interpolation is a method of constructing new points based on the existing ones. In other words finding a value of a new point in High Resolution (HR) image based on points in Low Resolution (LR) pictures.

In classical interpolation techniques pixels in LR data y can be related to the corresponding x values of HR data:

$$y_p = \frac{1}{N}\sum_{i=1}^{N} x_i + n, \text{ where}$$

y_p is the pixel of LR image at location p,
x_i is each one of the N High Resolution pixels contained within this LR pixel,
n is some additive noise.

In classical interpolation techniques the High Resolution pixels x are calculated as weighted average of existing LR pixels y.

$$x_i = \frac{1}{M}\sum_{i=1}^{M} w_{ij} * y_j, \text{ where}$$

w are weighted calculated as Euclidean distance between M existing surrounding LR pixels and the coordinates of new ones.

The biggest problem is that to find High Resolution data from the Low Resolution values. Unfortunately, there is an infinite number of values that meet that condition. Interpolation methods can be divided into three basic techniques.

The first group are the most common ones, like linear or spline-based interpolation. These techniques assume that it is possible to count the value of a new point by determination some kind of generic function. The main disadvantage is that they are correct only for images of homogeneous regions. As is well known, brain consists of grey substance, white substance and cerebral spinal fluid, so above-mentioned methods are not appropriate for MRI images interpolation. The second one is Super Resolution technique. It is commonly used to increase image resolution on functional MRI (fMRI) and Diffusion Tensor Imaging (DTI). The method is based on acquisition of multiple Low Resolution (LR) images of the same object. It is time consuming and not adequate for clinical applications.

The last but not least method is non-local patch-based technique which is based on self-similarity of a single image. It is possible to improve resolution by extracting information from a single image instead of acquiring several pictures.

First of all, there is a constraint which says that the original LR image has to be equal to the downsampled version X of the reconstructed image. It is named subsampling consistency constraint which is written as:

$$y_p - \frac{1}{N}\sum i = 1^N X_i = 0$$

To make the above equation real, the noise has to be removed. The presence of noise has to be minimalized on the LR image by applying a filter, for example MNLM3D filter for 3D MR images based on Non-Local Means filter. The filter removes the noise effectively without affecting the image structure.

The aim of this module is to increase MRI image resolution by upsampling. The input data is a single 320x240 image. After the processing it is going to be twice as big. To be more specific 640x480 pixels. The process can be named as double upsampling.

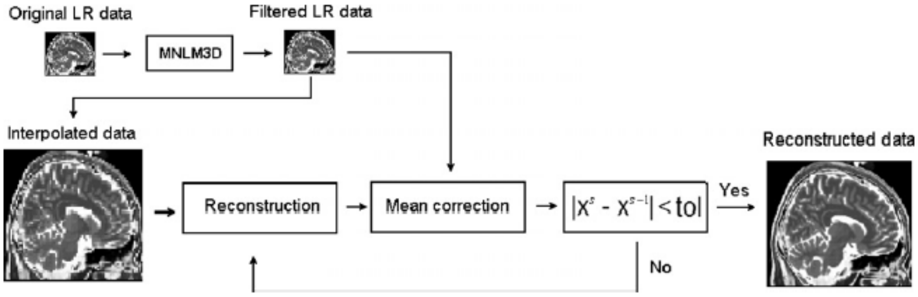
Original LR data
MNLM3D
Filtered LR data
Interpolated data
Reconstruction
Mean correction
$|x^{z} - x^{z-1}| < tol$
Yes
No
Reconstructed data

Figure 5.3. Diagram of the patch-based non-local algorithm [14]

.

In a nutshell, the first step of the algorithm is to divide every pixel of LR image into more pixels. Then the patch is designed. It is a rectangle which will be the area of interest. The pixels that are inside the area are taken into account during calculation the values of new points. After that an appropriate estimator is used to correctly calculate the values of new pixels. Every pixel has to be classified into one of three groups (grey substance, white substance or cerebral spinal fluid).

In order to show how the algorithm is designed the following pictured were generated using exemplary data. At the beginning there is an image 256x256 pixels.

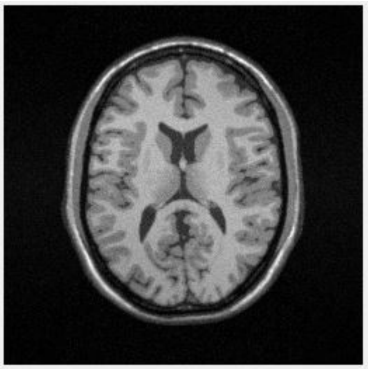

Figure 5.4. DICOM Low Resolution image 256x256

.

The first step is to add new pixels to increase the resolution M times horizontally and N times vertically. The pixels are equal to zero. After this process an appropriate estimator has to be chosen to make a decision about the values of new points.

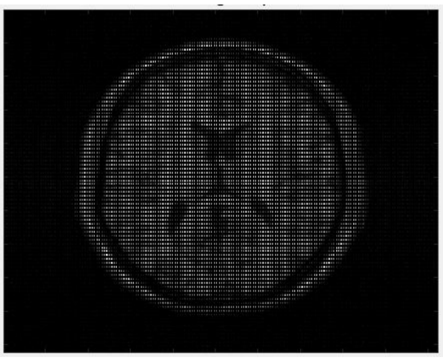

Figure 5.5. DICOM image with new points with N=2 and M=2

.

Patch-based non-local regularization is used to reconstruct new voxels. The voxel under study is reconstruced using all similar voxels which are placed inside the patch. Contrary to classical techniques which use local neighbourhood of the reconstructed voxel, the proposed method uses the only relevant information, the context of the voxel.

Another step in the algorithm diagram (Fig. 5.3) is mean correction. It has to be applied to ensure that the downsampled reconstructed HR X image is equal to the original y LR one. Thus the local mean value of X fits with the value of the y by adding the corresponding offset.

$$X = X + NN * (D(X) - y), \text{ where}$$

D is an operator that downsamples HR into original LR,
NN is a Nearest Neighbour operator that interpolates LR image to HR.

The Regularization and Correction steps are repeated until no important difference between two consecutive reconstructions. Tolerance is determined by Mean Absolute Difference .

List of References
[14] [**9art2**]

5.10. Module 11. Brain 3D

To prepare tree dimension visualization of the cerebral cortex algorithm of marching cubes is used.

The input data is multiple 2D slices of MR image. The marching cubes algorithm create a polygonal representation of constant density surfaces from a 3D array of data. To select the cerebral cortex is used output data from segmentation made in module 9. The structure of cortex is represented by value 3 in segmentation mask. The space of the image is divided into a regular grid of cubes. In each iteration one cube is considered. At each vertex of cube is determined how the surface intersects this cube. The density value are compared with the limit value - surface constant. If the data value is bigger than suface constant, one is assinged to a cube's vertex. There are 256 combinations of cube orientation relative to the surface, but we can distinguish 15 basic patterns, that repeat as symmetrical reflections, produces all possibilities (Fig. 5.6). If all values are less than the constant value, then the cube does not form any polygon. Otherwise, the edges of the polygon are defined (by linear interpolation) at the edges that intersect the surface. Using central differences, a unit normal at each cube vertex is calculated and then normal to each trangle vertex is interpolated. The output of the algorithm is the triangle vertices and vertex normals.

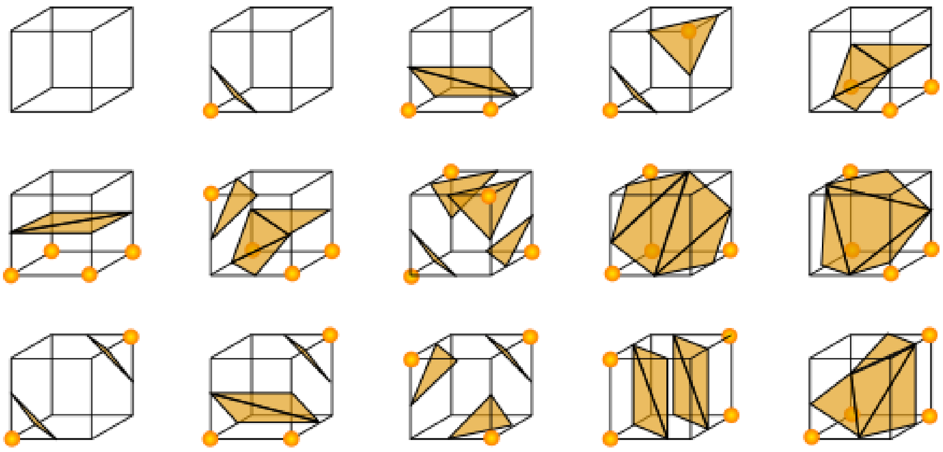

Figure 5.6. Triangulation for the 15 patterns.

To visualization the model, obtained by marching cubes, the VTK library is used, which enables building the three-dimension model. The VTK is object-oriented library. The classes of VTK is dedicated to processing and visualization data.

The second part of this module includes visualization of the brain's cross-section on arbitrarily defined plane. To enable selecting of intersection plane there was used object of VTK class, which allows set plane in elected direction by using computer mouse. When the plane is moved by user, in the real time the three-dimensional model is clipped in the place of selected plane. To improve quality of visualization the cross-section image, there is also possibility to see the image imposed on the three-dimensional model.

5.11. Module 12. Oblique imaging

The literature to this module is as useful as nipples on men. Everything is about inventing how to realize point 1. of the list.

Oblique Imaging is a technique to create non-perspective projections from 3D or multiple 2D images.

In order to create oblique image it is essential to:

- choose two angles under which the plane will be inclined,
- create a matrix of points that this plane consists of,
- from existing points pick those, which will be used in the image,
- interpolate points that are not existing.

Type of interpolation can vary, but in this project interpolation based on mean will be used. To interpolate one pixel mean of all pixels around him with given proximity is taken.

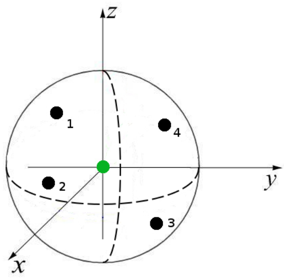

Figure 5.7. Visualization of pixels taken to interpolate

6. Implementation

6.1. Tools

Python - The main programming language of this project is Python. The utilized language version is **Python 3.5.4** (last "bugfix" release of Python 3.5).

VTK - The Visualization Toolkit (VTK) is an open-source software system for 3D computer graphics, modeling, image processing, volume rendering, scientific visualization, and information visualization. The project uses version **7.0**, as it is compatible with python 3.5 (compatibility assured by menpo project[1]).

Cython - Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language. It allows to represent the most computionally complex operations with more efficient code. This project uses version **0.26.1**.

PyQT - PyQt brings together the Qt C++ cross-platform application framework and the cross-platform interpreted language Python. This project uses its version **5.6** to create Graphical User Interface (GUI) for the application.

SciPy - SciPy is a collection of open source packages, which serves to import .mat MATLAB data into NumPy arrays and perform computations on them.

Conda - Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies. This project uses **Anaconda3** distribution to provide standarized enviroment with all of the required packages for our developers.

Pyinstaller - PyInstaller is a program that freezes (packages) Python programs into stand-alone executables. Despite of numerous package-specific problems, it appeared feasible to deploy the project using this tool.

6.2. Module 1. MRI reconstruction

Classical least squares estimation provides only the minimization of data error, which is an invalid solution. Conventionally, the LSE solution has a huge norm and thus it is a valueless outcome of a reconstruction procedure. To this end, we introduced basic Tikhonov regularization method that seem to partially overcome the problem. However, we firstly implemented LSE approach for later usage of the results obtained with this procedure in restoration of the images from a ‘nearby’ well-posed problem (regularization). For Tikhonov regularization, additional quadratic penalty term allows controlling the norm of the solution, which benefits in introduction of smoothness prior to estimated solution.

---

[1] http://www.menpo.org/

Firstly, we checked the data dimension that we load to the programme, as we could receive data starting from 3D to 5D. The implementation works for single slice structural and diffusion data (from many gradients) as well as for a whole set of brain slices. First two dimensions, provides information about data resolution i.e. 128×256, which means that the images where subsampled with factor $r = 2$. In case of structural data, the third dimension would be the number of slices (then, the forth is number of coils images) or simply number of coils images. For diffusion data, in 5D case the third dimension stands for number of slices, the forth - number of diffusion gradients and the fifth - number of coils images. The third dimension can be equal to one, so we get 4D data i.e. diffusion data for one slice. We also tested whether input dataset contains matrices with sensitivity maps profiles of a proper size, corresponding to maximal dimension of an image i.e. $256 \times 256 \times 8$, for case of images acquired with 8 coils. Furthermore, the input files should also contain subsampling factor r and number of coils L, if not, they can be calculated having the dataset's dimensional information.

Secondly, we performed 2D inverse Fourier Transform (2D IFT) on each of the dataset image, to transform them from **k**-space to **x**-space. This stage of an algorithm is presented in Fig. 6.1.

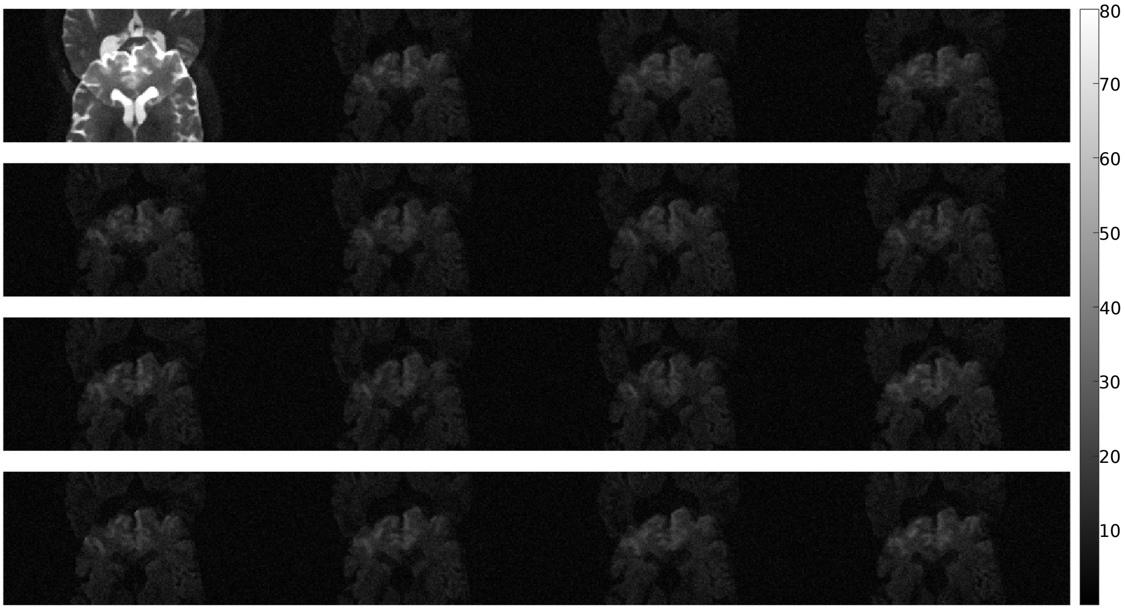

Figure 6.1. Subsampled diffusion dataset in **x**-space domain (data for one particular slice, acquired for 15 diffusion weightening gradients).

Thirdly, we used absolute value operator right before performing the SENSE reconstruction main step. As we are provided with sensitivity maps profiles (Fig. 6.2), we can easily reconstruct the subsampled data as it is shown in Fig. 6.3. The key idea is to evaluate the algorithm pixel by pixel. According to equation 5.4, the construction of defined $\mathbf{D}^S$ vector is simple. Basically, the $\mathbf{D}^S$ is a column vector of $L \times 1$ size, containing pixels withdrawn from each L coil image at specified spatial location. The matrix **S** we construct in a subsequent way: the $l - th$ row of $\mathbf{D}^S$ contains values of the sensitivity maps profiles, corresponding to data position in a subsampled image and moved by a subsampled FOV value (i.e. for $r = 4$, $FOV_y = 256/4 = 64$) $r - 1$ times. As a result, depending on the number of coils and subsampling rate value, we obtained a matrix of $L \times r$ size. In LSE sense, the result of computation of for defined $\mathbf{D}^S$ and **S** is column vector $\mathbf{D}^R$ ($r \times 1$ size).

However, the LSE approach is not an optimal solution, so we implemented the regularization approach which uses the *a priori* information of searched solution obtained with LSE algorithm. The images obtained after first

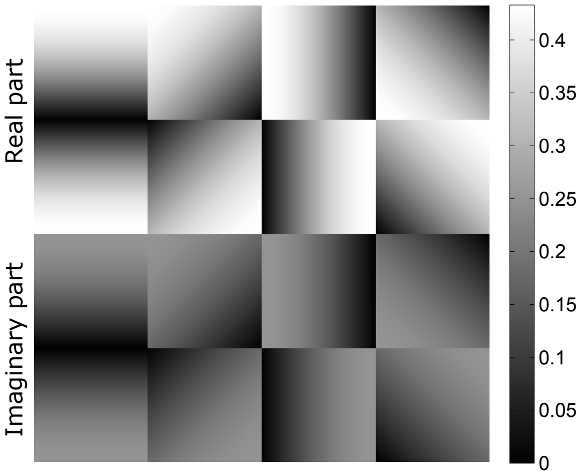

Figure 6.2. The real and imaginary parts of sensitivity maps used to reconstruct data ($L = 8$).

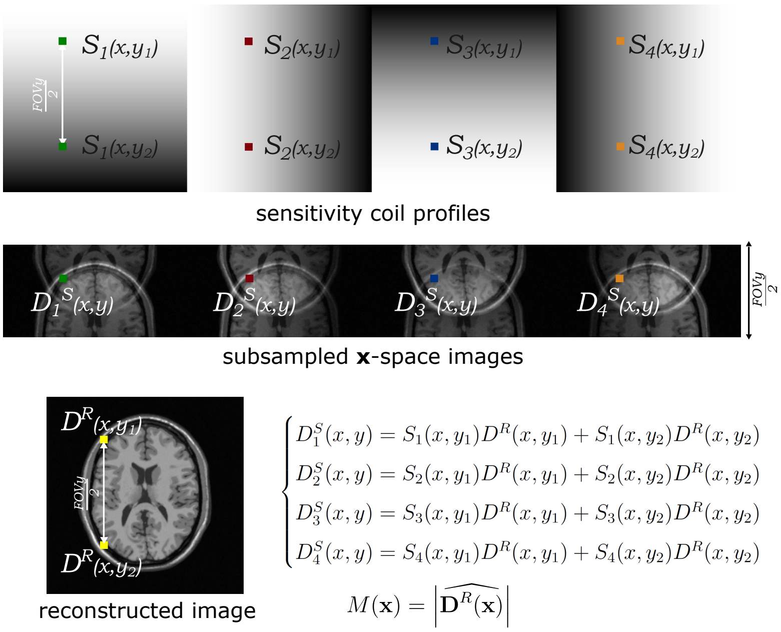
$S_1(x,y_1)$
$S_2(x,y_1)$
$S_3(x,y_1)$
$S_4(x,y_1)$
$\frac{FOVy}{2}$
$S_1(x,y_2)$
$S_2(x,y_2)$
$S_3(x,y_2)$
$S_4(x,y_2)$
sensitivity coil profiles
$D_1^S(x,y)$
$D_2^S(x,y)$
$D_3^S(x,y)$
$D_4^S(x,y)$
$\frac{FOVy}{2}$
subsampled **x**-space images
$D^R(x,y_1)$
$\frac{FOVy}{2}$
$D^R(x,y_2)$
reconstructed image
$D_1^S(x,y) = S_1(x,y_1)D^R(x,y_1) + S_1(x,y_2)D^R(x,y_2)$
$D_2^S(x,y) = S_2(x,y_1)D^R(x,y_1) + S_2(x,y_2)D^R(x,y_2)$
$D_3^S(x,y) = S_3(x,y_1)D^R(x,y_1) + S_3(x,y_2)D^R(x,y_2)$
$D_4^S(x,y) = S_4(x,y_1)D^R(x,y_1) + S_4(x,y_2)D^R(x,y_2)$
$M(\mathbf{x}) = \left|\widehat{\mathbf{D}^R(\mathbf{x})}\right|$

Figure 6.3. SENSE algorithm graphical explanation of Cartesian sampling using four receiver coils ($L = 4$) and the subsampling rate $r = 2$. Two yellow pixels of the reconstructed image are unfolded using coil sensitivity profiles and the corresponding folded pixels (points marked with green, red, blue and orange).

reconstruction are median filtered with window size3x3. We introduced this additional information to the solution of derived objective function according to Eq. 5.7. In this case, $\mathbf{D}^S$ and $\mathbf{S}$ are constructed in the same manner as for LSE case, however we regularized the solution with λ parameter. The choice of appropriate value of regularization parameter allows controlling the balance between both components (bias-variance tradeoff). For this scenario, we empirically picked λ parameter as a constant value. As the reconstruction is performed pointwisely, we introduced extra information about expected result as a vector $\mathbf{D}$ of rx1 size, containing pixels from reconstructed median filtered LSE images corresponding spatially to those to be reestimated with Tikhonov approach. The result is similar, i.e., we derived values of r reconstructed points, which differ from those obtained with basic LSE approach. Fig. 6.4 presents exemplary result of Tikhonov reconstruction algorithm implemented in the software.

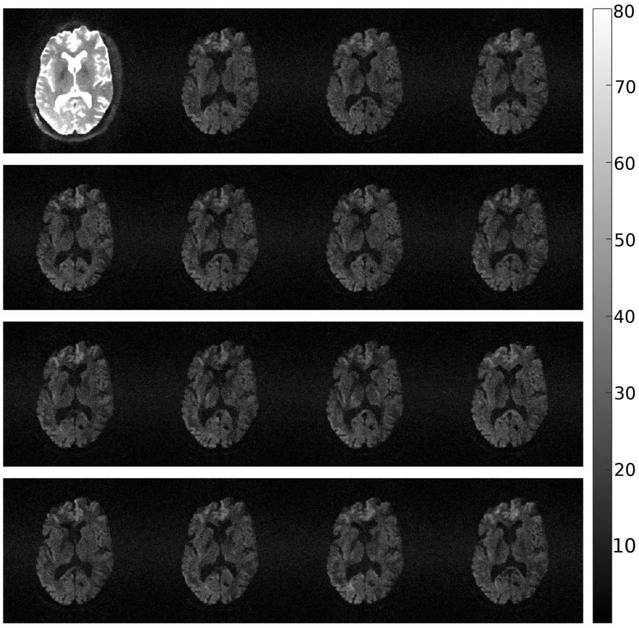

Figure 6.4. Reconstructed diffusion dataset in **x**-space domain (data for one particular slice, acquired for 15 diffusion weightening gradients).

6.3. Module 2. Intensity inhomogenity correction

-code

6.4. Module 3. Non-stationary noise estimation

-code

6.5. Module 4. Non-stationary noise filtering 1

-code

6.6. Module 5. Non-stationary noise filtering 2

Unbiased Non-Local Means algorithm was implemented based on [3]. Some, it is believed, improvements to original idea were introduced and described below. Implementation of the UNLM algorithm was divided into several steps:

- rewriting NLM prototype from MATLAB to Python,

- refactoring to UNLM,

- testing different Gaussian kernels,

- optimization of algorithm in order to reduce execution time,

- adjusting the algorithm to fit both types of data (structural and diffusion weighted).

Because some works and further improvements were done in parallel it is hard to describe implementation in chronological order.

Comparison of different Gaussian kernels

Once the main body of algorithm was ready, it was decided to run some tests to examine the influence of various Gaussian kernels on the output of the UNLM function. The main requirement for the Gaussian kernel was the fact that the overall sum of all elements within kernel should be equal to 1. Another, for sake of sanity, requirement built on the shape of the kernel surface - the slope should be gentle in order to avoid ’overfitting’ to the central pixel.

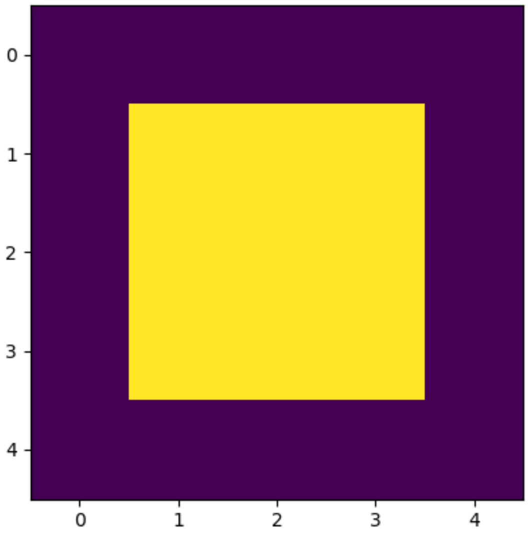

Figure 6.5. First tested Gaussian kernel.

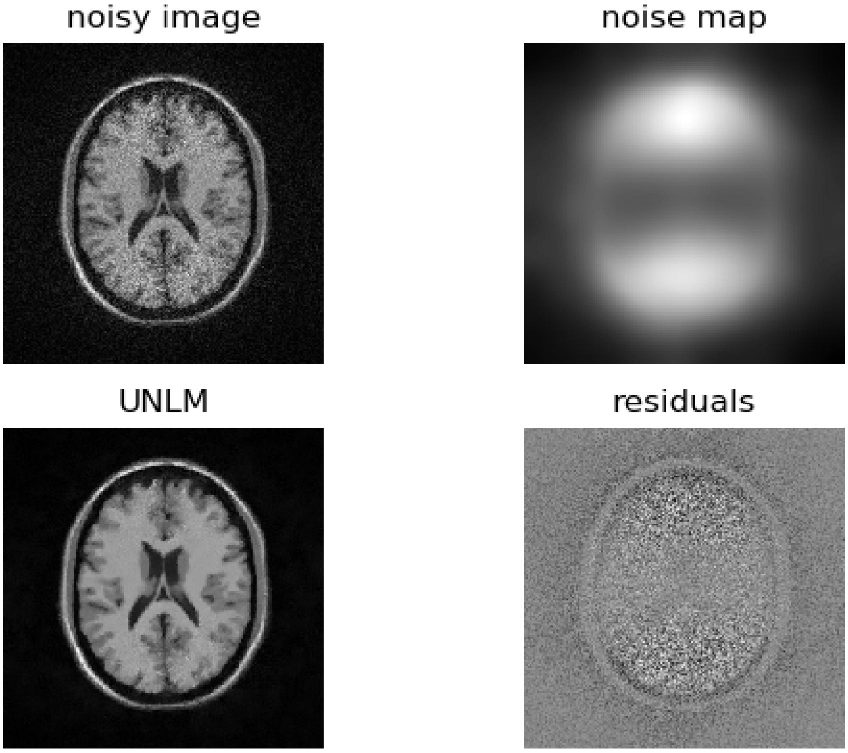

Figure 6.6. Results obtained for first kernel.

First kernel was presented in the article describing the algorithm [3]. However, kernel was considered as not optimal due to its two-level shape. To overcome the problem of wrong shape another attempt took place. Second generated kernel had desired shape, yet values didn't meet the requirement of summing to 1.

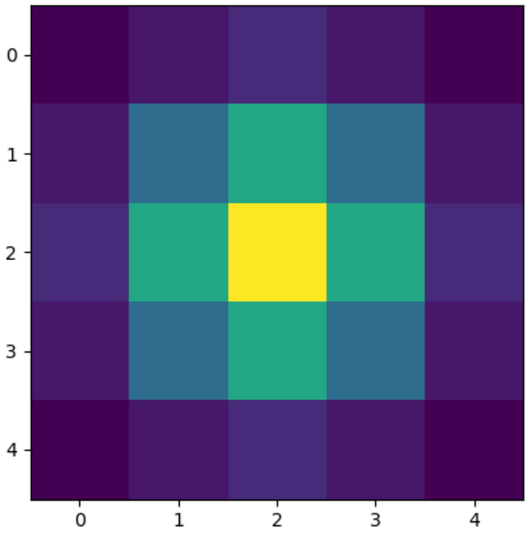

Figure 6.7. Second tested Gaussian kernel.

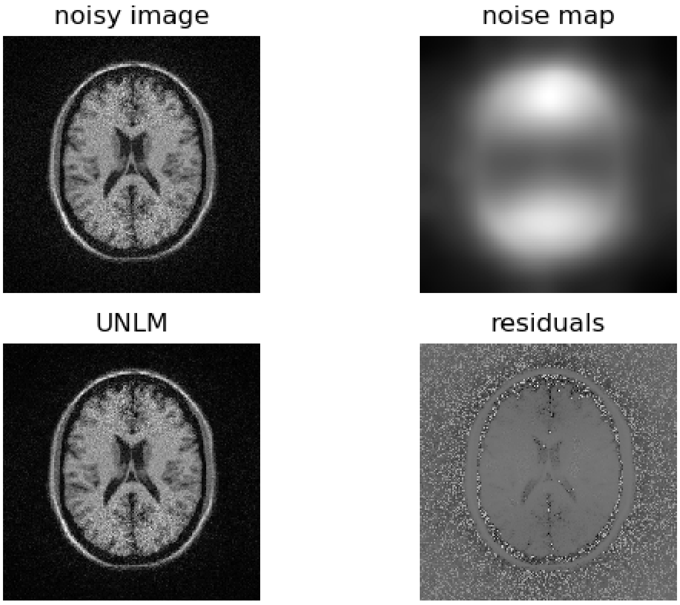
noisy image
noise map
UNLM
residuals

Figure 6.8. Results obtained for second kernel.

Third kernel, again was disappointing. It had a shape of first kernel, but represented different values, since it was generated in other manner.

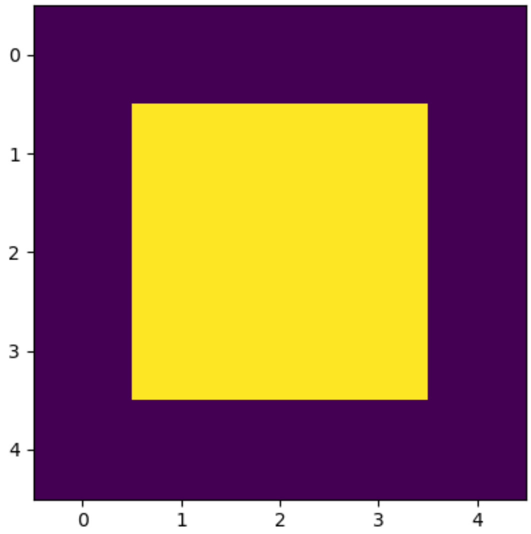

Figure 6.9. Third tested Gaussian kernel.

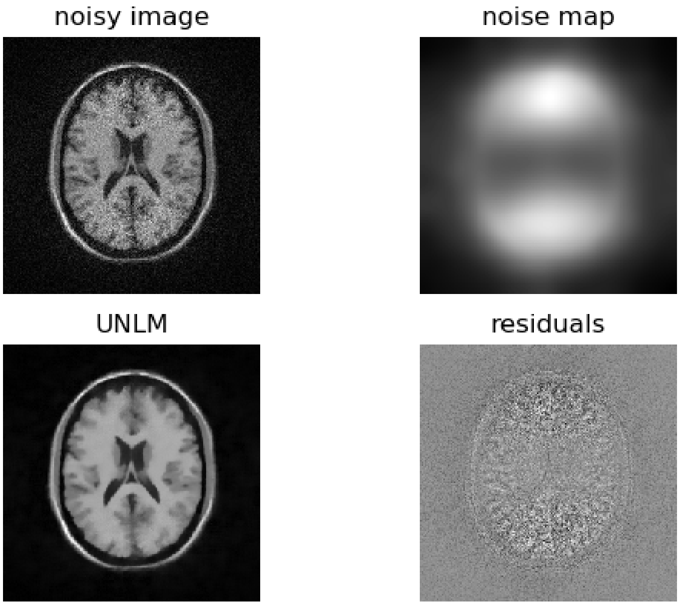
noisy image
noise map
UNLM
residuals

Figure 6.10. Results obtained for third kernel.

Fourth kernel, had all properties that were desired. It was decided to use this particular one in implementation of the UNLM algorithm.

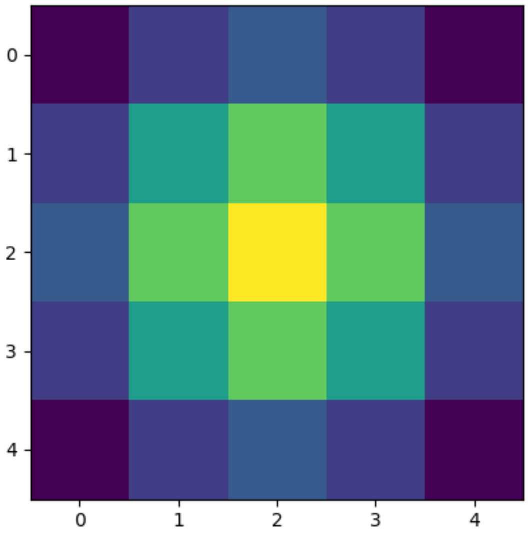

Figure 6.11. Fourth, and final, tested Gaussian kernel.

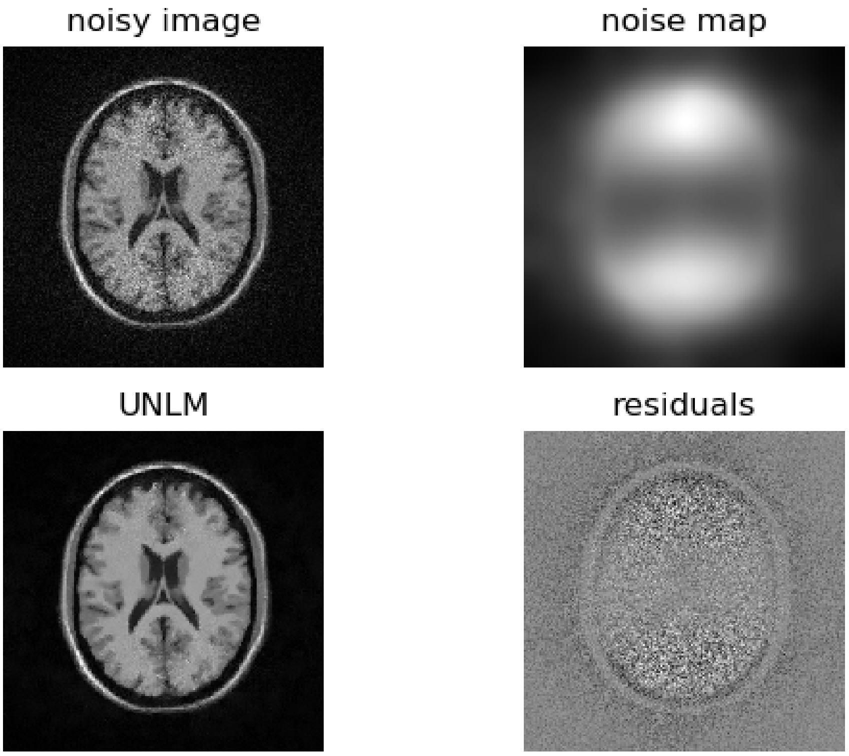
noisy image
noise map
UNLM
residuals

Figure 6.12. Results obtained for fourth kernel.

Someone with an eye for detail can spot the differences between results achieved whilst using all kernels. Most eye-catching distinction lies in boundaries of the brain and skull.

Following code snippet presents the implementation of Gaussian window, that was finally chosen after running some tests. This particular window has a special characteristic of being shallow - the central pixel's value doesn't stand out from values in close vicinity.

```
def gk4(rsim):

# function that prepares Gaussian window used to penalize
# pixels based on distance within window
kerlen = 2 * rsim + 1
nsig = 2
interval = (2*nsig+1.)/kerlen
x = np.linspace(-nsig-interval/2., nsig+interval/2., kerlen+1)
kern1d = np.diff(st.norm.cdf(x))
kernel_raw = np.sqrt(np.outer(kern1d, kern1d))
kernel = kernel_raw/kernel_raw.sum()
return kernel
```

Listing 6.1. Code used for Gaussian kernel generation.

Reducing execution time

After line-by-line translation of code from MATLAB to Python filtration of single slice lasted for 180 seconds. That was unacceptable, so few flaws in the code had to be identified. It's worth mentioning that single slice filtering in MATLAB took 19 seconds on average.

The first faced incorrectness was the fact of not fully using the numpy library utilities. Changing 'pure python' functions like *sum* to numpy equivalents allowed reduction of time to 80 seconds per slice. That run time was still insufficient. After code profiling with *cProfile* library it was identified, that nested for loops used to move around image were most time consuming. As a result it was decided to get rid off loops and change tedious loop-based architecture to multi-dimensional matrices operations. That operation resulted in achieving 3.5 seconds on average. Involving matrices operations allows to fully exploit numpy implementation (which itself is based on C, so it is faster than Python).

Furthermore, an attempt to use threads in order to speed up calculations on multi-dimensional data took place. For that test, a synthetic 3D data was generated. Processing each slice in a for loop takes 3.5 x *number of slices* seconds. Involving threads should reduce the time, but unfortunately it was doubled. So a tough decision of processing slices in classic loop has been made. Other possibilities of parallel computing were not taken into consideration due to the compatibility issues with GUI.

Unbiased Non-Local Means algorithm

Having reduced the computation time and having chosen suitable penalty function (Gaussian kernel) algorithm presents as in code snippet below.

```
def unlm(image, n_map):

# setting window sizes to reduce number of parameters and chance failure caused by user
rsearch = 5
rsim = 2
```

```
# get size for indexes
[m, n] = image.shape

# extend image at boundaries to make calculations on edges of image possible
img_ext = np.pad(image, rsim, 'symmetric')

# create penalty function
penalty = np.asarray(gk4(rsim))

# set space for filtered image
output = np.zeros([m, n])

# precompute possible window2
window2_precomp = np.zeros((m, n, rsearch, rsearch))
for i in range(2, m):
for j in range(2, n):
window2_precomp[i, j, :, :] = img_ext[i - rsim:i + rsim + 1, j - rsim:j + rsim + 1]

# p pixel loop
for i in range(0, m):
for j in range(0, n):
ii = i + rsim
jj = j + rsim
window1 = np.asarray(img_ext[ii - rsim:ii + rsim + 1, jj - rsim:jj + rsim + 1])

# limit the search space
pmin = max(ii - rsearch -1, rsim)
pmax = min(ii + rsearch - 1, m)
qmin = max(jj - rsearch -1, rsim)
qmax = min(jj + rsearch - 1, n)

# define placeholders for values
w_max = 0
avg = 0
w_sum = 0

# set exponential decay
h_sq = 1 / np.square(1.22 * n_map[i, j])

window2_slice = window2_precomp[pmin:pmax, qmin:qmax, :]
window_diff = window1[None, None, :, :] - window2_slice
d = np.sum(penalty[None, None, :, :] * (window_diff * window_diff), axis=(2, 3))

w = np.exp((-1)*d * h_sq)

# primitive way of handling case: pixel p == pixel q
w[w == 1] = -100
w_max = np.amax(w)
w[w == -100] = w_max

w_sum = np.sum(w, axis=(0, 1))
avg = np.sum(w * img_ext[pmin:pmax, qmin:qmax], axis=(0, 1))
avg = avg + w_max * img_ext[ii, jj]
w_sum = w_sum + w_max

if w_sum > 0:
value = avg / w_sum
output[i, j] = abs(cmath.sqrt(value * value - 2 * (np.square(n_map[i, j]))))
else:
output[i, j] = image[i, j]

return output
```

Listing 6.2. UNLM algorithm code.

To explain the code and present it in more picturesque way, one can look at the image filtering as moving set of 2D windows on image and calculating some metrics, according to their positions and values. There are two types of windows: neighbourhood and search space. Size of neighbourhood window is indirectly defined by rsim parameter, which was assigned by default to 2. That results in neighbourhood window of size 5x5, becasue 5 = 2 * rsim +1. Search space window is dependent ot rsearch parameter, which was assigned to 5 giving 11x11 window. The values of these parameters were set based on analysis conducted in [3]. Authors discovered that these sizes give best results in filtration.

Neighbourhood windows are centered around pixels p and q, which are pixel being filtered and filtering pixel correspondingly. Figure below presents the idea of moving windows.

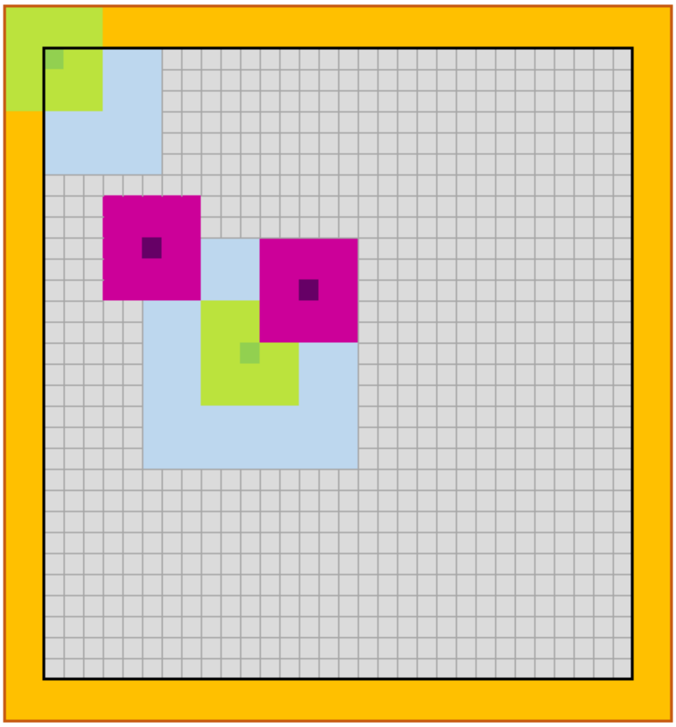

Figure 6.13. Graphical representation of moving windows.

Given figure has to be explained. Gray area represents 2D picture, the grid on in resembles pixels. Dark green square (pixel) is the p pixel and the light green area around it is the neighbourhood window defined by rsim. Similarly, purple area and dark purple pixel resembles window of pixel q. The size of the image has to be extended by rsim in each direction to make calculations on edges of original image possible. This scenario is presented in the top left corner of picture. Pixel p is within area of image but its window doesn't fit into original image. As a consequence of extending the image, the orange area is temporarily added to the image. The values in the added space are assigned by symmetrical padding near edges. Last thing that has to be deciphered is blue square. It is the search space window. It narrows the possibilities of q pixel positions taken into consideration whilst calculating distances between p and q windows.

It is worth mentioning that there exist special case, when q pixel is in the same position as p pixel. If that situation would have been treated equally to other, what would introduce some bias towards 'self-distance'. According to the equation used to calculate distance (d) between windows, one has to calculate element-wise difference of windows. Having both windows in the same position would result in distance equal to 0, which further can have negative influence on the output. To overcome this problem, maximum weight found within the window is assigned to a weight calculated based on 'zero' distance.

Adjusting the algorithm for both types of data

Denoising using UNLM method is used by both data types, structural and diffusion weighted. There are some differences in implementation of the algorithm for both types, however the same code is used for them in this module. It is caused by analysis of diagrams presented in [6]. They showed that taking gradient information gives no significant information in filtering result. On this occasion, the only difference in handling both types of data lays in main function of the module. More precisely, after making decision which data is being processed loops work on different dimensions. Following code snippet presents the *run_module* function, where data is being accurately handled.

```
def run_module(mri_input, other_arguments=None):

if isinstance(mri_input, smns.mri_diff):
[m, n, slices, grad] = mri_input.diffusion_data.shape
data_out = np.zeros([m, n, slices, grad])

for i in range(slices):
for j in range(grad):
data_out[:, :, i, j] = unlm(mri_input.diffusion_data[:, :, i, j], mri_input.noise_map[:, :, i,
    j])

mri_input.diffusion_data = data_out

elif isinstance(mri_input, smns.mri_struct):
[m, n, slices] = mri_input.structural_data.shape
data_out = np.zeros([m, n, slices])

for i in range(slices):
data_out[:, :, i] = unlm(mri_input.structural_data[:, :, i], mri_input.noise_map[:, :, i])

mri_input.structural_data = data_out

else:
return "Unexpected␣data␣format␣in␣module␣number␣5!"
return mri_input
```

Listing 6.3. run_module function.

One decision must be finally justified. The whole bigger product is targeted (theoretically) at physicians that usually have no knowledge of image filtering. On this occasion it was decided to reduce number of parameters responsible for algorithm outcome to 0. That solution should reduce possibilities of getting badly-processed data, as optimal parameters are set by default.

6.7. Module 6. Diffusion tensor imaging

Preprocessing and Module I/O

In order to improve diffusion tensor estimation it is imperative to remove artifacts. In addition to standard MRI pre-processing, one needs to correct for artifacts arising the use of diffusion-gradient pulse sequences and longer acquisition time. While hardware manufacturers try to proactively diminish some of these effects, software processing is still mandatory.

Module Input:

- 3D structural data array of shape X x Y x Z, where XY - pixel image intensities, Z - chosen slice, which is the T1- or T2-weighted image corresponding the the given DWI acquisition
- 4D diffusion data array of shape X x Y x Z x M, where XY - pixel image intensities, Z - chosen slice, M - applied diffusion gradient direction
- b_value, a scalar value corresponding to applied diffusion gradient sequence magnitude
- 2D gradients matrix of shape M x 3, where each row corresponds to a normalized (x, y, z) components of diffusion gradient sequence vectors
- optionally - 3D binary mask of shape X x Y x Z, corresponding to the brain area detected by Module 8 (Skull Stripping); if not supplied, DTI is computed on each input data voxel

Module Output:

- list of size Z, corresponding to each slice; every list element is a dictionary of biomarker images: MD, RA, FA, VR of shape X x Y, and biomarker FA_rgb of shape X x Y x 3

6.7.1. Initialization

In order to abstract DTI implementation from end-user, all classes and methods other than the main function `run_module` are private to module source code script. It is important to note that prior to running the module one has to provide the module with input data object, as well as SOLVER and FIX_METHOD parameters. SOLVER passed as an argument decides whether to use WLS or NLS estimation, whole FIX_METHOD decides how to "fix" negative eigenvalues.

As mentioned in the detailed description chapter, 'ABS' takes absolute value of each eigenvalue, while 'CHOLESKY' ensures that the estimated tensor is positive definite. Eigenvalues of positive definite matrices are always non-negative. 'ABS' is a post-estimation fix, meaning that it does not modify the default estimation algorithm (i.e. it is applied after WLS or NLS computation), while 'CHOLESKY' directly modifies the expressions for WLS and NLS cost function gradients and Hessian matrices.

After passing all required arguments to the `run_module` function, they are reshaped internally in order to be compatible with module. Concretely, `DTISolver` class instance, computing the DTI proper, assumes that input data argument is a concatenated 3D array of both structural and diffusion images, which are stored separately in the original data structure. Moreover, b_value and gradient fields are reshaped to be lists correpsonding to each slice of the new data array (that is: b_value is repeated in length while both have zeros appended that correspond to structural images). Finally, all of the above is done separately for each slice and DTI module performs it's computation slice-by-slice due to memory constraints.

6.7.2. WLS estimation

WLS with the ABS fix method is a fast yet simple method of module pipeline computation based on diffusion tensor estimation. As such these parameters were set as default for DTI.

Diffusion tensor estimate was computed by implementing the equation:

$$\gamma = \left(\boldsymbol{W}^T \boldsymbol{\omega}^T \boldsymbol{\omega} \boldsymbol{W}\right)^{-1} \boldsymbol{W}^T \boldsymbol{\omega}^T \boldsymbol{\omega} \boldsymbol{y} \tag{6.1}$$

using NumPy matrix broadcasting operations, effectively abstracting away array reshaping. Weights vector $\boldsymbol{\omega}$ is calculated using a separate function in order to avoid changing every piece of code refering to WLS weights in case they change. The following implementation assumes the simplest of models presented in the Detail Description chapter, that is weights being equal to the measured signal.

6.7.3. NLS estimation

In case of NLS estimation, in addition to implementing gradient and Hessian matrix computation methods:

$$\begin{aligned} \nabla f_{NLS} &= -\boldsymbol{W}^T \hat{\boldsymbol{S}} \boldsymbol{r} \\ \nabla^2 f_{NLS} &= \boldsymbol{W}^T \left(\hat{\boldsymbol{S}}^{\boldsymbol{T}} \hat{\boldsymbol{S}} - \boldsymbol{R} \hat{\boldsymbol{S}}\right) \boldsymbol{W} \end{aligned} \tag{6.2}$$

It is important to devise an iterative scheme because gradient result depends on NLS diffusion tensor estimate. For that reason an algorithm based on [9] has been implemented. The method itself is called a Modified Newton's Algorithm and can be summarised as in Fig.6.14.

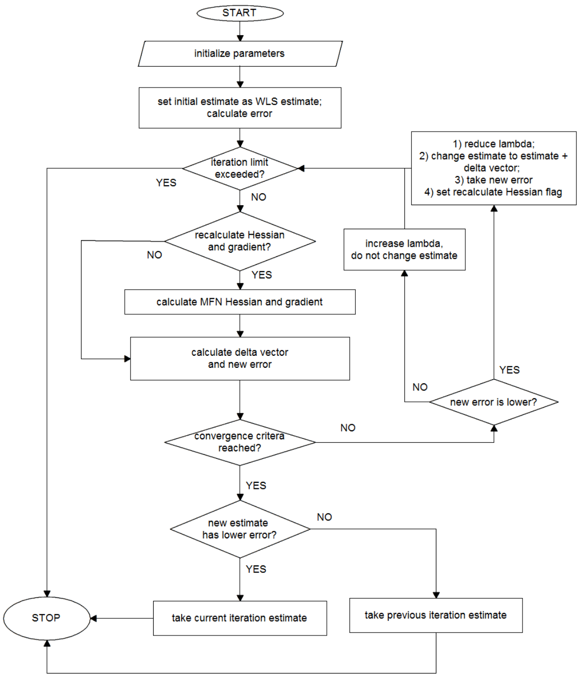
START
initialize parameters
set initial estimate as WLS estimate;
calculate error
iteration limit
exceeded?
YES
NO
recalculate Hessian
and gradient?
NO
YES
calculate MFN Hessian and gradient
calculate delta vector
and new error
convergence critera
reached?
NO
YES
new estimate
has lower error?
NO
YES
1) reduce lambda;
2) change estimate to estimate +
delta vector;
3) take new error
4) set recalculate Hessian flag
increase lambda,
do not change estimate
YES
NO
new error is lower?
STOP
take current iteration estimate
take previous iteration estimate

Figure 6.14. Modified Newton's method for iterative computation of NLS estimate (based on [9])

.

The following parameters (collectively known in code as MFN parameters) were set:

- MFN_MAX_ITER = 3 - iteration limit
- MFN_ERROR_EPSILON = 1e-5 - first convergence criterion (error change is small)
- MFN_GRADIENT_EPSILON = 1e-5 - second convergence criterion (vanishing gradient)
- MFN_LAMBDA_MATRIX_FUN = ’identity’- regularization matrix added to Hessian matrix
- MFN_LAMBDA_PARAM_INIT = 1e-4 - initial regularization matrix multiplier

Delta estimate is calculated using the following formula:

$$\delta = -\left(\nabla^2 f_{NLS} + \lambda I\right)^{-1} \nabla f_{NLS} \tag{6.3}$$

with λ parameter increasing and decreasing by a factor of 10, depending on whether newly calculated estimate yields lower error (decreasing for lower error, increasing otherwise).

Convergence is established using the following formulas:

$$\begin{aligned} |f_{NLS_{new}} - f_{NLS_{new}}| &< \texttt{MFN\_ERROR\_EPSILON} \\ \delta^T \nabla f_{NLS} &< \texttt{MFN\_GRADIENT\_EPSILON} \end{aligned} \tag{6.4}$$

6.7.4. Biomarkers computation

As mentioned previously, the estimate obtained from NLS or WLS methods can be reshaped to a 3x3 matrix:

$$\boldsymbol{D} = \begin{bmatrix} D_{xx} & D_{xy} & D_{xz} \\ D_{yx} & D_{yy} & D_{yz} \\ D_{zx} & D_{zy} & D_{zz} \end{bmatrix} \tag{6.5}$$

assuming our estimate is equivalent to:

$$\boldsymbol{\gamma} = [lnS_0, D_{xx}, D_{yy}, D_{zz}, D_{xy}, D_{yx}, D_{xz}]^T \tag{6.6}$$

Tensor estimation results for a sample 126x126x55 slice are presented on Fig.6.15.

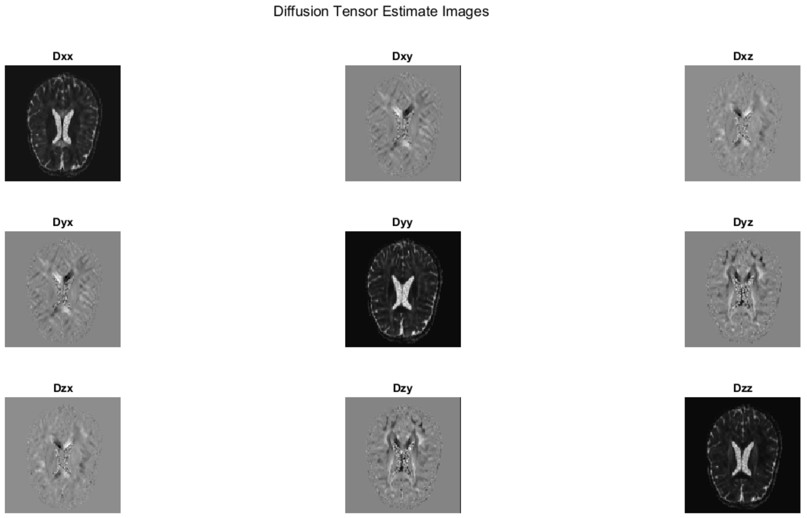

Figure 6.15. Diffusion tensor estimate using the WLS-ABS method for a 126x126x55 test image.

We can then compute the eigenvalue decomposition of $\boldsymbol{D}$ and obtain eigenvalues and eigenvectors for each pixel of input image. Eigenvalues are sorted in descending order and saved for later computation (Fig.6.16). Moreover, eigenvectors corresponding to the highest eigenvalue are saved to separate variable, since they represent the direction of principal diffusion.

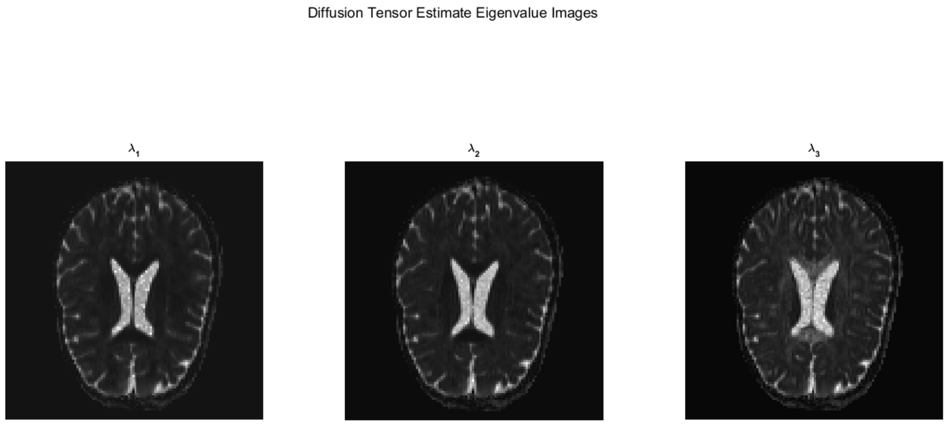

Figure 6.16. Diffusion tensor estimate eigenvalues of sample imgae; eigenvalues sorted in descending order from left to right.

Biomarkers are then computed using the following formulas:

$$MD = \frac{\lambda_1 + \lambda_2 + \lambda_3}{3} \tag{6.7}$$

$$RA = \sqrt{\frac{(\lambda_1 - MD)^2 + (\lambda_2 - MD)^2 + (\lambda_3 - MD)^2}{3\,MD}} \tag{6.8}$$

$$FA = \sqrt{\frac{3}{2}}\sqrt{\frac{(\lambda_1 - MD)^2 + (\lambda_2 - MD)^2 + (\lambda_3 - MD)^2}{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}} \tag{6.9}$$

$$VR = \frac{\lambda_1 \lambda_2 \lambda_3}{MD^3} \tag{6.10}$$

Computed biomarkers of the sample image are presented on Fig.6.17. It is important to note that skull is visible because the brain area was selected by hand instead of relying on Module 08 output.

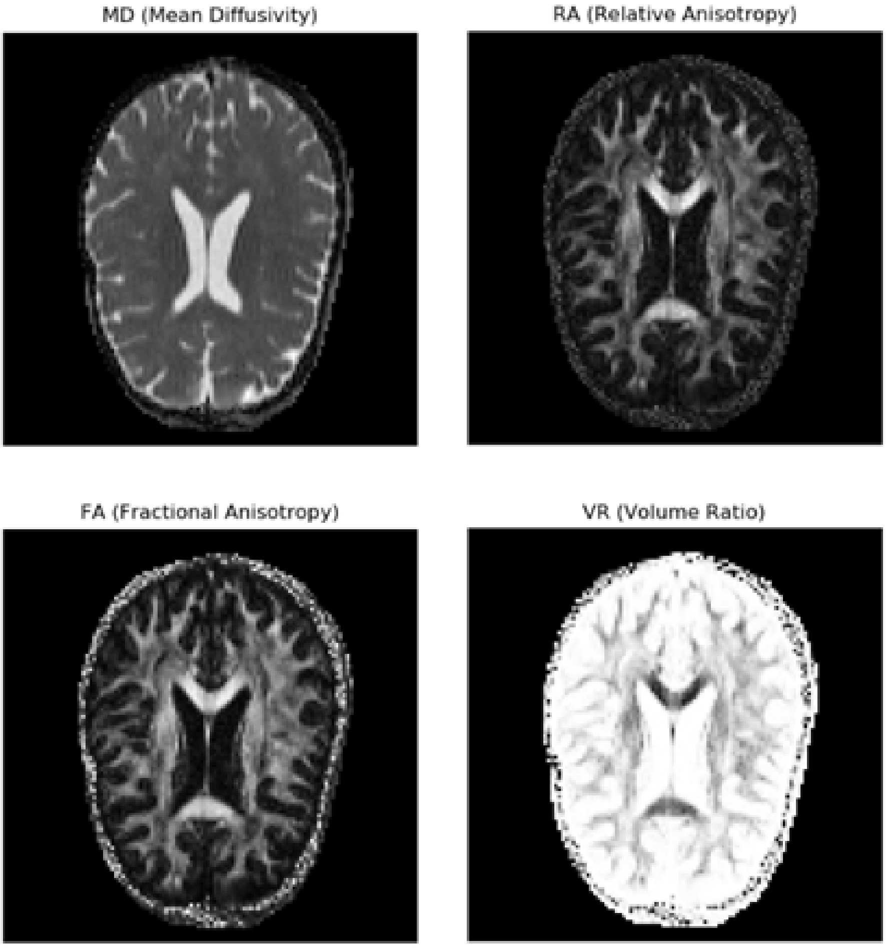
MD (Mean Diffusivity)
RA (Relative Anisotropy)
FA (Fractional Anisotropy)
VR (Volume Ratio)

Figure 6.17. Diffusion biomarkers of sample image. Top left: MD, top right: RA, bottom left: FA, bottom right: VR.

Figure 6.18 presents the fifth biomarker image present in output dictionary - FA-weighted principal direction of diffusion color map. Color-encoded direction for this slice is presented as follows:

- red - transversal (left-right)
- green - anterior-posterior (front-back)
- blue - cranio-caudal (head-feet)

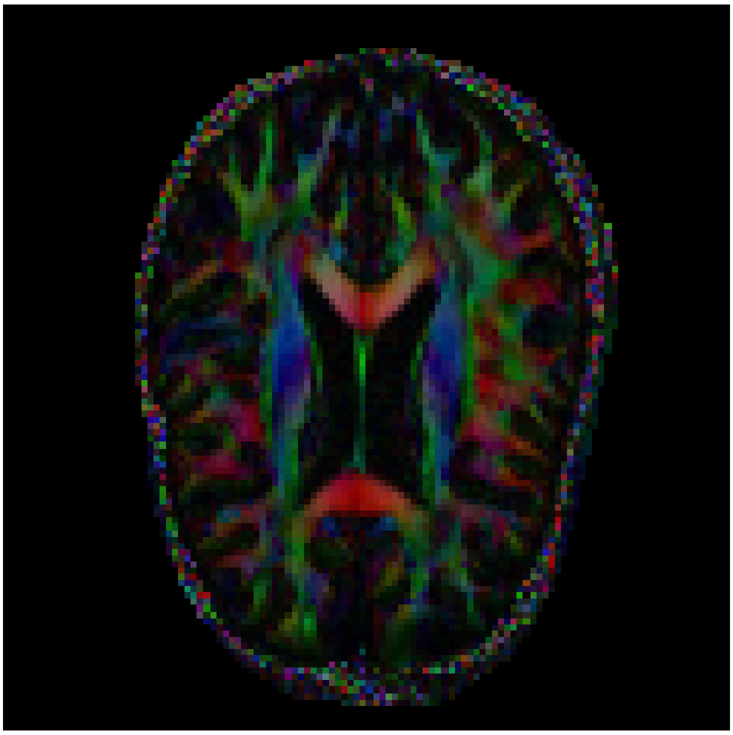

Figure 6.18. FA-weighted principal direction of diffusion of sample image.

6.7.5. Implementation caveats

Module source code was implemented and tested on a single sample image outlined above. It constituted a perfect basis for algorithm evaluation because it was already preprocessed, without the need to wait for other modules implementation. Even though it lacked a skull stripping mask, hand-drawn shape was good enough for testing, even though skull outline is visibly present, particularily on Fig.6.18.

It was difficult to determine which estimation method - WLS or NLS - is objectively better. WLS was significantly faster and it was not much different from NLS results. This might have been due to the fact, that the diffusion-encoding gradient matrix was unusually large (7 structural images and 48 DWIs). Testing on preprocessed raw data, supplied for application evaluation has shown differences between methods, although both estimates are valid (to the Author's best knowledge) as it is difficult to compare any estimation methods without reference.

Initially developed using nested for-loops, DTI module has been vectorized to speed up computations. For a supplied 256x256x16 DWI stack a single slice is being processed in less than 3 seconds for WLS-ABS, while other methods scale linearily with chosen number of iterations (starting at 6 seconds, then 3 seconds per MFN iteration). It must be noted that the module was not tested with skull stripping mask, which could have drastically reduced computation time. Moreover computation time heavily depends on hardware constraints.

6.8. Module 8. Skull stripping

-code

6.9. Module 9. Segmentation

-code

6.10. Module 10. Upsampling

The implementation began with making a decision about the input data. The input image has to be 2 dimensional without a noise. The uploaded image will be considered as a 2 dimensional array of double values. Other parameters that will be needed are vertical and horizontal extensions. It has to be a total number. The size of the input array will be multiplied by the number. For example, the image 256x256 pixels with extensions equal to 3 will be 768x768 pixels as an output. The function needs also the window size. In one iteration, the function will take into consideration only pixels witch are inside the window. The loop will stop only when the whole image will be interpolated. To see the results the *plotting* parameter has to be a boolean *True* value.
The below picture shows the block diagram of the algorithm (Fig. 6.19) with comments added.

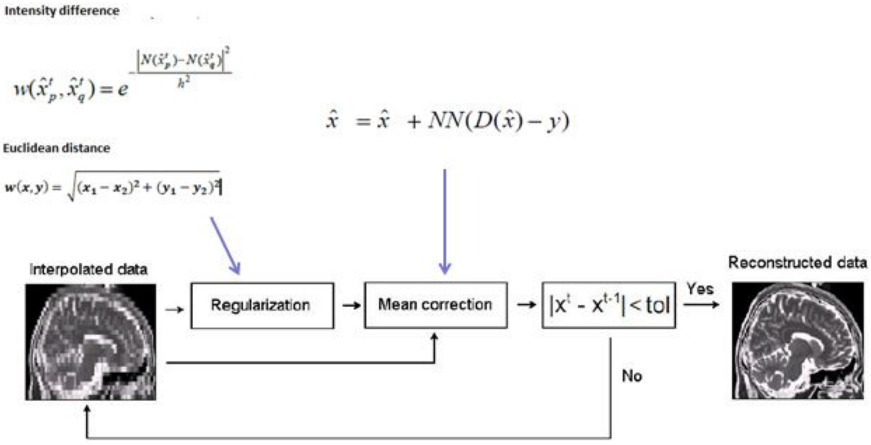
Intensity difference
$w(\hat{x}_p^t, \hat{x}_q^t) = e^{-\frac{|N(\hat{x}_p^t)-N(\hat{x}_q^t)|^2}{h^2}}$
$\hat{x} = \hat{x} + NN(D(\hat{x}) - y)$
Euclidean distance
$w(x,y) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
Interpolated data
Regularization
Mean correction
$|x^t - x^{t-1}| < tol$
Yes
No
Reconstructed data

Figure 6.19. Algorithm block diagram with comments

.

The following steps of the implementation will be discussed:

1. **Initial interpolation**
   To form the input image into the output with chosen size the initial interpolation is needed. As showed in [14] the spline interpolation is used. In short, spline interpolation is an interpolation where interpolant is a special kind of piecewise polynomial called spline. It is recommended, because of the fact that the method makes small error even with low degree polynomials for the spline. Taking into consideration the edges the extrapolation was applied. To be more specific, the last row *n* is equal to the previous one *n-1*. The same with the first row, the first column and the last column of the input. To show the results, the extensions are set as 2. The input data is 256x256 pixels. The undermentioned picture (Fig. 6.20) shows the result of the initial interpolation. The scale is included so it is easy to see that the size is twice as big. Now, the zeros (Fig. 5.5) have nonzero values, so the next steps can be made.

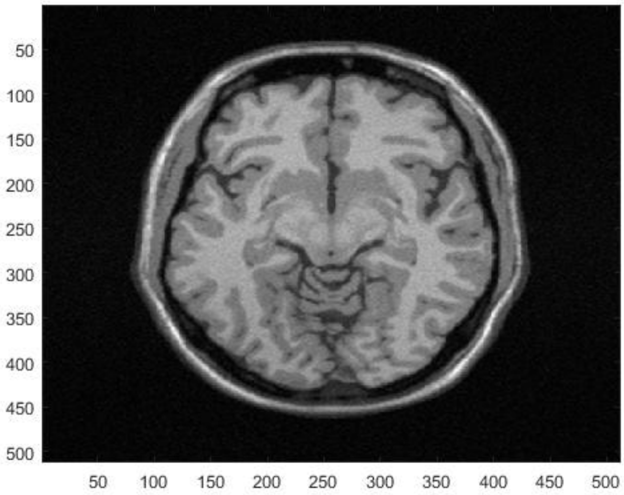

Figure 6.20. The image after initial interpolation 512x512 pixels

.

2. **Regularization**

   The main functionality is the regularization step. The function makes the square window which contains some pixels. In one iteration of a loop, only the pixels which are inside the window are considered. Next, the window moves and takes another pixels. It is repeated till the end of the image. The main goal is to determine of the weight which will be used at the end.

   The first weight is calculated as the difference between the pixels intensities.

   $$w_1(x, y) = e^{\frac{|N(x)-N(y)|^2}{h^2}}, \text{ where}$$

   $N(x)$, $N(y)$ are window and image intensity values,

   h is a level, which is equal to the half of the standard deviation of the input image.

   The next part of the whole weight is an Euclidean distance weight. It is simply calculated as the Euclidean distance between every pixel in the window and every pixel in the image. It is based on the below equation.

   $$w_2(x, y) = \frac{\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}}{h^2}, \text{ where}$$

   h is a level, which is equal to the half of the standard deviation of the input image.

   The total weight is calculated as $w_{total} = w_1 * w_2$. The output image is determined as the result of the sum of the multiplication of the weight and the window, divided by the sum of the weight.

3. **Mean correction**

   The mean correction step is a place in the function where the correctness of the equation $X = X + NN * (D(X) - y)$. It was abovementioned, that the downsampled HR has to be equal to the input data.

4. **Tolerance checking** At the beginning it was established that only the 2D images will be taken into consideration and the upsampling function will be applied only once, so the step is excluded. There were experiments to upsample the data several times, but the result were not satisfied.

6.11. Module 11. Brain 3D

The implementation of 11th module included visualization of cortex structure in three-dimensional model and visualization of elected cross-section of model.

As input parameter, module get ouput of 9th module. This is the mask of segmentated data, which includes cortex structure as value 3. The data about cortex structure is seperated from 3D array. 11 th module is displayed in new window of application, so implementation includes not only functionality requirements but also designing of graphical user interface.

To prepare design of user interface used Qt Designer program.

Module is selecting by user in main window of application. If input data is correct, new window is opened and reconstrucion of three-dimensional model is initialized automatically.

First input data – 3D array – is converted to vtkFloatArray, which makes vtkImageData object. Then basing on vtkImageData there is used vtkMarcingCubes class, which makes reconstruction. As a thresholds there was used value equal to 1, because of fact that all values not equal to zero are included to cortex structure.

After reconstrucion model is displayed in BRAIN 3D window. To enable displaying VTK objects in Qt Application there is set frame in which is inserted the object of VTK library by using QVTKRenderWindowInteractor. It is dedicated vtkWidget to display VTK in QT ibrary.

To visualization data there are used following classes:

- vtkRenderer – which enables rendering process: transforming geometry, light and camera view into an image
- vtkMapper – which maps data to graphics primitives
- vtkActor – adjusting data to 3D scenery
- vtkInteractorStyleTrackballCamera – setting possibility of interaction with model

BRAIN 3D window enables three options:

- preview 3D model
- clip model
- clip model and show plane.

Automatically after initalization there is loaded firts mode: preview 3D model. Mode: clip model, enables to setting intersection plane and cut model in place of it. Intersection plane is vtkImagePlaneWidget object. It allows to interactive set the plane by computer mouse. Interaction of plane is activated whenever user press „clip model" button. When interaction event is detected the model is automatically clip by vtkCliPolyData class.

Mode: clip model and show plane call the same function as previous mode, but with input parameter planemode - True (default – False). It expands functionality of diplaying the cross-section corresponding to the elected plane. It improves readability of intersection plane. It is based on the same objects of VTK library, but using additional methods of it.

User has possibility fluently switch modeLs by pressing appropriate buttons. Application has also help window, with short user guide.

6.12. Module 12. Oblique imaging

-code

7. Tests

7.1. Module 1. MRI reconstruction

-tests for module 1

7.2. Module 2. Intensity inhomogenity correction

7.3. Module 3. Non-stationary noise estimation

7.4. Module 4. Non-stationary noise filtering 1

7.5. Module 5. Non-stationary noise filtering 2

7.6. Module 6. Diffusion tensor imaging

In order to verify whether any changes to the module source code result in module no longer working properly, a series of unit tests have been designed and implemented. This automates the procedure of testing module functionality after every change. This addition resulted not only in validating functionality, but also allowed to detect unexpected behaviour and edge cases, which lead to changes in source code.

7.6.1. High-level tests

Methods of the `Module06Tests` class are designed to verify high-level functionality of DTI module. Specifically, it allows to check whether the resulting biomarkers are calculated as expected. All of the following methods are tested for each solver-fix method combination (that is: WLS-ABS, WLS-CHOLESKY, NLS-ABS, NLS-CHOLESKY).

Interface Tests

This group of methods test module behaviour when receiving incorrect data from GUI:

- `test_invalid_input_data` - checks whether module raises error when input data structure strutural data field is empty
- `test_invalid_solver_type` - checks whether module raises error when input solver is neither ’WLS’ nor ’NLS’

- `test_invalid_fix_method` - checks whether module raises error when input fix method is neither 'ABS' nor 'CHOLESKY'

Data Object Tests

This group of methods test whether input data instance is modified during module execution:

- `test_object_modified_in_place` - tests whether module output is a view of input data memory instead of a new object (it is in order to be less memory-intensive)
- `test_input_biomarker_field_changed` - verifies if input object biomarkers field is modified during module execution
- `test_input_other_fields_not_changed` - compliments the aforementioned methods by checking whether all fields but biomarkers where changed during module execution or not

Output Tests

This group of methods check the properties of resulting biomarkers field:

- `test_biomarker_fields_check` - verifies whether all expected biomarkers (MD, RA, FA, VR, FA_rgb) have been calculated during module execution and written to input data class instance
- `test_biomarkers_for_each_slice` - tests whether biomarkers dictionary is calculated for each slice present in input data and returned as Python list of dictionaries
- `test_biomarker_shape` - validates the shape of biomarker images (whether XY dimensions are the same as input data as well as whether they are gray or in color)

7.6.2. Low-level tests

Methods of the `DTISolverTests` class are designed to verify low-level functionality of DTI computation by testing the behaviour of class constructor of DTISolver and verifying methods called during pipeline module execution abstracted from the end user.

Constructor Tests

This group verifies class constructor for valid and invalid input information:

- `test_solver_object_created` - verifies the output of class constructor for valid input data
- `test_input_invalid_bvalue` - checks whether an error is raised when constructor is presented with invalid (empty) b_value
- `test_input_invalid_gradients` - checks whether an error is raised when constructor is presented with invalid (empty) gradient matrix
- `test_input_invalid_data` - checks whether an error is raised when constructor is presented with invalid (empty) data array

- `test_input_negatiove_data` - checks whether an error is raised when constructor is presented with invalid (negative values) data array

Skull Stripping Mask Tests

This group of methods verifies solver class object behaviour when presented with invalid skull stripping masks. Since this input information is optional, instead of raising an error in response to invalid mask, a default mask (all pixels in image are valid for computation) is used during execution:

- `test_input_skull_mask_empty` - verifies that the default mask is used if input mask is empty

- `test_input_invalid_mask` - verifies that the default mask is used if input mask is not binary

Eigenvalue Computation Tests

This group of methods the sign of eigenvalues with and without 'ABS' or 'CHOLESKY' fix. For proper biomarker camputation it is important to have non-negative diffusion tensor eigenvalues.

- `test_eigenvalues_abs_fix` - compares eigenvalue computation with 'ABS' fix method (expected output: non-negative eigenvalues when using the fix method, no guarantee otherwise)

- `test_eigenvalues_cholesky_fix` - compares eigenvalue computation with 'CHOLESKY' fix method (expected output: non-negative eigenvalues when using the fix method, no guarantee otherwise)

Biomarker Computation Tests

This last group contains only one test method, designed to check module behaviour when computing biomarkers:

- `test_negative_eigenvalue_biomarker_warning` - checks whether a warning is issued during biomarker computation if input eigenvalues are negative (no error is raised)

7.7. Module 8. Skull stripping

7.8. Module 9. Segmentation

7.9. Module 10. Upsampling

Results

To visualize the upsampling functionality several output with different parameters are showed (Fig. 7.1, 7.2).

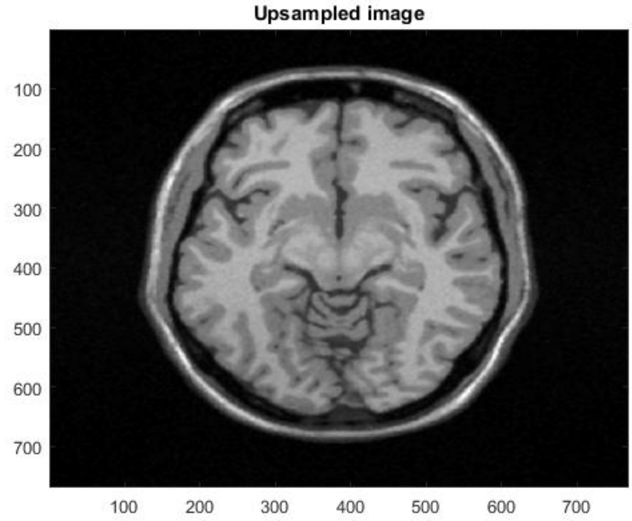
Upsampled image
100
200
300
400
500
600
700
100
200
300
400
500
600
700

Figure 7.1. The upsampled image with extensions equal to 3 and window equal to 2

.

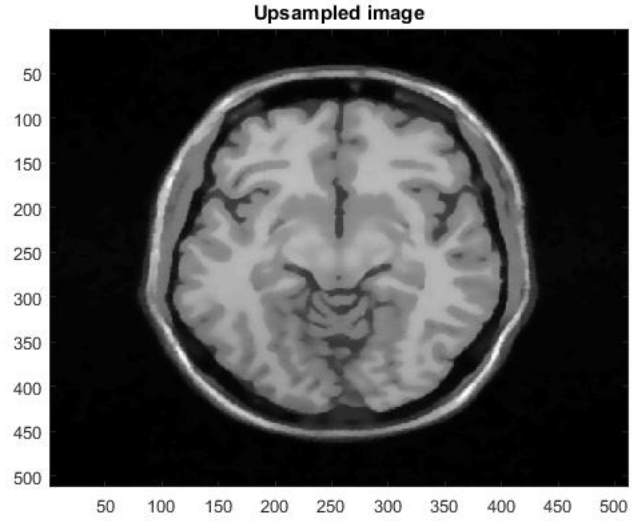
Upsampled image
50
100
150
200
250
300
350
400
450
500
50
100
150
200
250
300
350
400
450
500

Figure 7.2. The upsampled image with extensions equal to 2 and window equal to 5

.

To better see the comparison (Fig.7.3, 7.4)

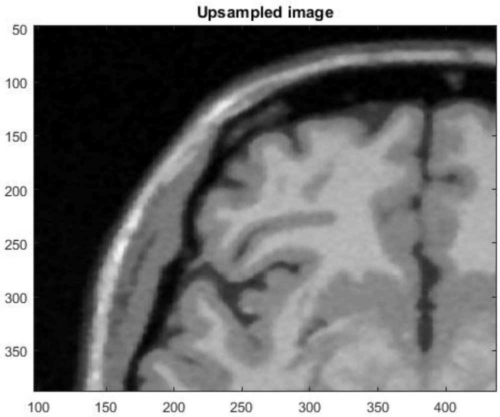
Upsampled image
50
100
150
200
250
300
350
100
150
200
250
300
350
400

Figure 7.3. The upsampled image with extensions equal to 3 and window equal to 2

.

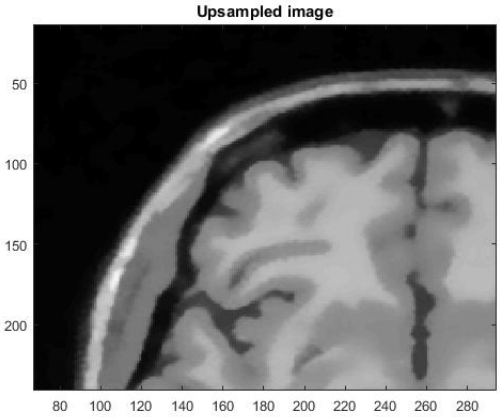
Upsampled image
50
100
150
200
80
100
120
140
160
180
200
220
240
260
280

Figure 7.4. The upsampled image with extensions equal to 2 and window equal to 5

.

Comparison with the other methods

To confirm the better functionality of the proposed method the results will be compared. The other are Linear, Nearest Neighbour, Cubic and Spline interpolation methods (Fig 7.5, 7.6, 7.7, 7.8). To better see the comparison (Fig. 7.9).

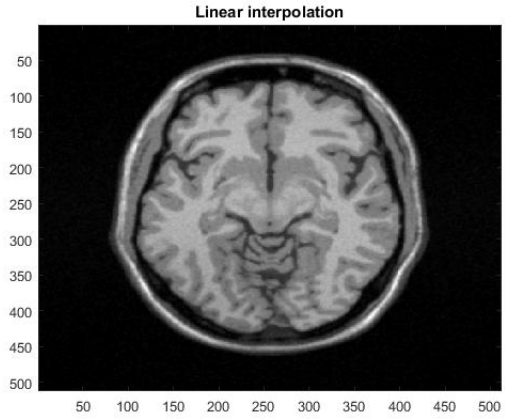
Linear interpolation
50
100
150
200
250
300
350
400
450
500
50 100 150 200 250 300 350 400 450 500

Figure 7.5. Linear interpolation

.

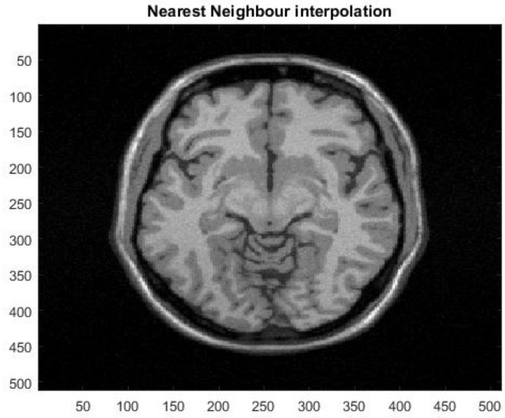
Nearest Neighbour interpolation
50
100
150
200
250
300
350
400
450
500
50 100 150 200 250 300 350 400 450 500

Figure 7.6. Nearest neighbour interpolation

.

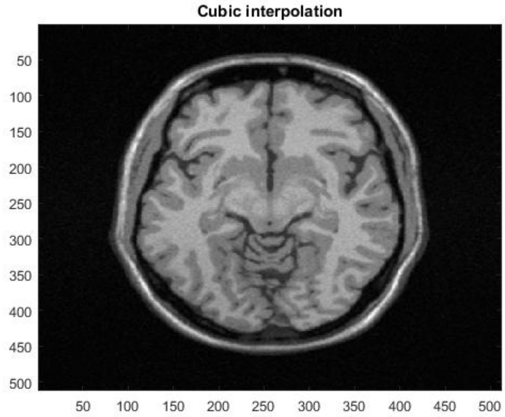
Cubic interpolation
50
100
150
200
250
300
350
400
450
500
50 100 150 200 250 300 350 400 450 500

Figure 7.7. Cubic interpolation

.

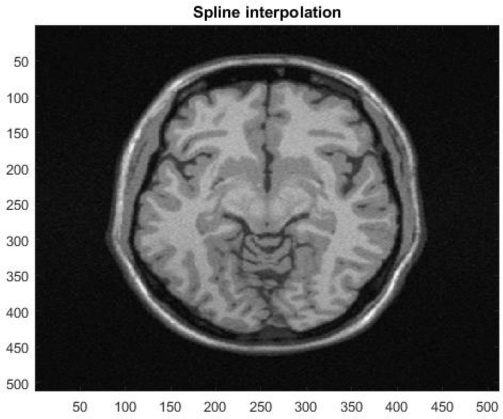
Spline interpolation
50
100
150
200
250
300
350
400
450
500
50
100
150
200
250
300
350
400
450
500

Figure 7.8. Spline interpolation

.

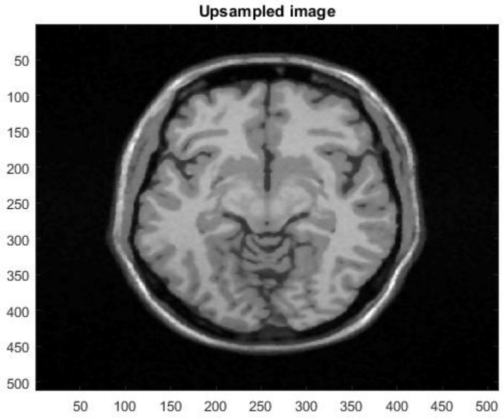
Upsampled image
50
100
150
200
250
300
350
400
450
500
50
100
150
200
250
300
350
400
450
500

Figure 7.9. Proposed method

.

Conclusions

In conclusion, implementation of a new interpolation method met with success. Unquestionably, the proposed method gives better results than other popular interpolation methods.

7.10. Module 11. Brain 3D

7.11. Module 12. Oblique imaging

7.12. Application

-whole app tests

8. Authors

Authors of this project are students of Biomedical Engineering, AGH UST, Krakow, Poland.

Name	Role
Sylwia Mól	Project Manager
Jacek Fidos	Software architect
Maciej Gryczan	GUI engineer
Adrian Stopiak	Vizualization engineer
Malwina Molendowska	1st module developer
Klaudia Gugulska	2nd module developer
Kacper Turek	3rd module developer
Magdalena Rychlik	4th module developer
Alicja Martinek	5th module developer
Mateusz Pabian	6th module developer
Anna Grzywa	8th module developer
Magdalena Kucharska	9th module developer
Eliza Kowalczyk	10th module developer
Karolina Gajewska	11th module developer
Michał Kotarba	12th module developer

List of Figures

3.1 Dependencies between modules 12

5.1 Triangulation for the 15 patterns. 28
5.2 Triangulation for the 15 patterns. 29
5.3 Diagram of the patch-based non-local algorithm [14] 31
5.4 DICOM Low Resolution image 256x256 31
5.5 DICOM image with new points with N=2 and M=2 32
5.6 Triangulation for the 15 patterns. 33
5.7 Visualization of pixels taken to interpolate 34

6.1 Subsampled diffusion dataset in **x**-space domain (data for one particular slice, acquired for 15 diffusion weightening gradients). 36
6.2 The real and imaginary parts of sensitivity maps used to reconstruct data ($L = 8$). 37
6.3 SENSE algorithm graphical explanation of Cartesian sampling using four receiver coils ($L = 4$) and the subsampling rate $r = 2$. Two yellow pixels of the reconstructed image are unfolded using coil sensitivity profiles and the corresponding folded pixels (points marked with green, red, blue and orange). 37
6.4 Reconstructed diffusion dataset in **x**-space domain (data for one particular slice, acquired for 15 diffusion weightening gradients). 38
6.5 First tested Gaussian kernel. 39
6.6 Results obtained for first kernel. 40
6.7 Second tested Gaussian kernel. 41
6.8 Results obtained for second kernel. 41
6.9 Third tested Gaussian kernel. 42
6.10 Results obtained for third kernel. 42
6.11 Fourth, and final, tested Gaussian kernel. 43
6.12 Results obtained for fourth kernel. 43
6.13 Graphical representation of moving windows. 46
6.14 Modified Newton's method for iterative computation of NLS estimate (based on [9]) 50
6.15 Diffusion tensor estimate using the WLS-ABS method for a 126x126x55 test image. 51
6.16 Diffusion tensor estimate eigenvalues of sample imgae; eigenvalues sorted in descending order from left to right. 52

6.17 Diffusion biomarkers of sample image. Top left: MD, top right: RA, bottom left: FA, bottom right: VR. 53
6.18 FA-weighted principal direction of diffusion of sample image. 54
6.19 Algorithm block diagram with comments 55
6.20 The image after initial interpolation 512x512 pixels 56

7.1 The upsampled image with extensions equal to 3 and window equal to 2 62
7.2 The upsampled image with extensions equal to 2 and window equal to 5 62
7.3 The upsampled image with extensions equal to 3 and window equal to 2 63
7.4 The upsampled image with extensions equal to 2 and window equal to 5 63
7.5 Linear interpolation 64
7.6 Nearest neighbour interpolation 64
7.7 Cubic interpolation 64
7.8 Spline interpolation 65
7.9 Proposed method 65

Bibliography

[1] Zujun Hou. “A Review on MR Image Intensity Inhomogeneity Correction”. In: *International Journal of Biomedical Imaging* 2006 (2006), pp. 1–11. DOI: *10.1155/IJBI/2006/49515*.

[2] Uro Vovk, Franjo Pernus, and Botjan Likar. “A Review of Methods for Correction of Intensity Inhomogeneity in MRI”. In: *IEEE Transactions on Medical Imaging* 26 (2007), pp. 405–421. DOI: *10.1109/TMI.2006.891486*.

[3] J MANJON et al. “MRI denoising using Non-Local Means”. In: *Medical Image Analysis* 12.4 (2008), pp. 514–523. DOI: *10.1016/j.media.2008.02.004*.

[4] R.D. Nowak. “Wavelet-based Rician noise removal for magnetic resonance imaging”. In: *IEEE Transactions on Image Processing* 8.10 (1999), pp. 1408–1419. DOI: *10.1109/83.791966*.

[5] Nobuyuki Otsu. “A Threshold Selection Method from Gray-Level Histograms”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66. DOI: *10.1109/tsmc.1979.4310076*.

[6] Antonio Tristán-Vega and Santiago Aja-Fernández. “DWI filtering using joint information for DTI and HARDI”. In: *Medical Image Analysis* 14.2 (2010), pp. 205–218. DOI: *10.1016/j.media.2009.11.001*.

[7] Jose Soares et al. “A hitchhiker’s guide to diffusion tensor imaging”. In: *Frontiers in Neuroscience* 7 (2013), p. 31. ISSN: 1662-453X. DOI: *10.3389/fnins.2013.00031*.

[8] Cheng Guan Koay et al. “Investigation of anomalous estimates of tensor-derived quantities in diffusion tensor imaging”. In: *Magnetic Resonance in Medicine* 55.4 (2006), pp. 930–936. ISSN: 1522-2594. DOI: *10.1002/mrm.20832*.

[9] Cheng Guan Koay et al. “A unifying theoretical and algorithmic framework for least squares methods of estimation in diffusion tensor imaging”. In: *Journal of Magnetic Resonance* 182.1 (2006), pp. 115 –125. ISSN: 1090-7807. DOI: *10.1016/j.jmr.2006.06.020*.

[10] Raymond Salvador et al. “Formal characterization and extension of the linearized diffusion tensor model”. In: *Human Brain Mapping* 24.2 (2005), pp. 144–155. ISSN: 1097-0193. DOI: *10.1002/hbm.20076*.

[11] Peter Basser, J Mattiello, and D LeBihan. “Estimation of the Effective Self-Diffusion Tensor from the NMR Spin Echo”. In: 103 (Apr. 1994), pp. 247–54. DOI: *10.1006/jmrb.1994.1037*.

[12] Peter J. Basser and Derek K. Jones. “Diffusion-tensor MRI: theory, experimental design and data analysis – a technical review”. In: *NMR in Biomedicine* 15.7-8 (2002), pp. 456–467. ISSN: 1099-1492. DOI: *10.1002/nbm.783*.

[13] Bart Goossens Ivana Despotović and Wilfried Philips. “MRI Segmentation of the Human Brain: Challenges, Methods, and Applications”. In: *Hindawi Publishing Corporation* 2015 (2015). DOI: *10.1155/2015/450341*.

[14] Jose V. Manjon et al. “Non-Local MRI Upsampling”. In: *Elsevier* (2010). DOI: *10.1016/j.media.2010.05.010*.