VFA T1 Mapping using Derivative Free Optimization techniques

Yashika Patil
University of Massachusetts Dartmouth

April 21st, 2024

VFA (Variable Flip Angle) T1 mapping is a widely used technique in quantitative MRI for measuring tissue relaxation times, with applications ranging from basic research to clinical diagnosis. In this study, we focus on the computation time required by the derivative-free optimization methods, NOVIFAST, Nelder Mead, and Implicit Filtering to converge to a solution. The experiments were conducted on a dataset consisting of images with dimensions 256x256 pixels and varying slice thicknesses. Our results demonstrate that NOVIFAST exhibits the fastest computation time among the three methods. On the other hand, Nelder-mead and Implicit-filtering take slightly longer computation times. These findings provide valuable insights into the selection and optimization of algorithms for VFA T1 mapping, facilitating their application in clinical and research settings.
Keywords: VFA T1 mapping, Relaxation times, NOVIFAST, Nelder Mead, Implicit filtering.

1 Introduction

Medical imaging is a crucial task in healthcare, allowing doctors to see inside the body without surgery. Among these methods, Magnetic Resonance Imaging (MRI) is exceptional for its ability to scan various body areas, providing clear images of soft tissues. It uses a strong magnetic field and radiofrequency (RF) pulses to generate these detailed images. When a patient enters the MRI scanner, the protons in the body align with the magnetic field created by the scanner. A brief burst of RF energy is applied (RF excitation), which temporarily disrupts the alignment of the protons. After the pulse is turned off, the protons begin to realign with the magnetic field. In this process, they emit energy in the form of radiofrequency signals. The specialized coils in the MRI scanner detect the emitted radiofrequency signals. These signals are then processed by a computer to create detailed images of the internal structures of the body. Relaxation time is the time taken by disrupted protons to return to their original alignment. There are two main types of relaxation processes: T1 relaxation (longitudinal relaxation) and T2 relaxation (traverse relaxation). The RF pulses are sent multiple times to generate both T1-weighted and T2-weighted images, providing different contrasts and insights into various tissues.

In T1 mapping, the focus is on quantifying the T1 relaxation time of tissues. This process involves acquiring a series of images using different flip angles during the RF excitation phase of the MRI scan. By varying the flip angle, which determines the amount of energy imparted to the protons, different tissues exhibit varying signal intensities. Through mathematical modeling and analysis of these signal intensities, T1 values can be estimated for each voxel within the scanned region.

VFA (Variable Flip Angle) T1 mapping is a particular method within T1 mapping where the flip angle is adjusted for each acquisition to optimize the accuracy of T1 measurements. This technique allows for more precise determination of T1 relaxation times, enhancing the ability to differentiate between different tissue types and detect subtle variations in tissue properties.

Accurate T1 mapping holds significant importance in both clinical practice and research settings. It provides crucial information about tissue composition, structure, and function, offering insights into various physiological and pathological conditions. Clinically, T1 mapping is used for diagnosing diseases, monitoring treatment responses, and guiding interventions. Research applications include investigating tissue properties, understanding disease mechanisms, and developing new

imaging biomarkers for early disease detection.
In this study, we attempt the implementation of the derivative-free optimization algorithms, namely, Nelder-Mead and Implicit-filtering for VFA T1 mapping.

2 Background

T1 mapping techniques rely on acquiring multiple images with different flip angles and fitting their signal intensities to a mathematical model to estimate T1 values. The VFA method is commonly used for T1 mapping, where a series of images are acquired with varying flip angles, typically between 2° and 20°. The relationship between flip angle, repetition time (TR), and tissue T1 relaxation time can be described by the Ernst equation, facilitating T1 estimation.
Traditional optimization techniques, such as least squares fitting, are often used to estimate T1 values from VFA data. However, these methods may suffer from sensitivity to noise, susceptibility to local minima, and computational inefficiency. To overcome these limitations, advanced optimization algorithms, such as NOVIFAST, Nelder-Mead, and Implicit Filtering, have been proposed.

2.1 SPGR Model

The SPGR (Spoiled Gradient Recalled Echo) model is a mathematical component in magnetic resonance imaging (MRI) for T1 mapping, a technique used to quantify tissue properties. In MRI sequences, the SPGR method involves acquiring a series of MR images with varying flip angles (FAs) but constant repetition time (TR) and echo time (TE). The SPGR signal model describes the relationship between the observed signal intensity (y) and parameters such as the scaling factor (K), longitudinal relaxation time ($T1$), TR, and flip angles (α). Specifically, the SPGR signal model is given by:

$$s_n(K, T1) = \frac{K(1 - e^{\frac{-TR}{T1}}) \sin(\alpha_n)}{1 - e^{\frac{-TR}{T1}} \cos \alpha_n} \tag{1}$$

The above equation can be rewritten as:

$$s_n(K, T1) = \frac{c_1 \sin(\alpha_n}{1 - c_2 \cos(\alpha_n)} \tag{2}$$

where $c_1 = K(1 - e^{-\frac{TR}{T1}})$, and $c_2 = e^{-\frac{TR}{T1}}$.
s_n represents the SPGR signal intensity for the nth image, where $n = 1 \dots N$, α_n denotes the flip for each image, and N is the total number of images acquired. The parameter K accounts for factors such as the longitudinal component of the net nuclear magnetization vector and attenuation due to $T2^*$ relaxation.

The VFA (Variable Flip Angle) SPGR method, allows for T1 mapping by fitting the SPGR signal model to the acquired MR images. The estimation of $T1$ relaxation time involves voxel-wise fitting of the SPGR model to the MR images, which provides insights into tissue composition and structure.

2.2 NOVIFAST

Non-linear least squares (NLLS) estimation is a fundamental problem and has applications in medical imaging. In the context of variable flip angle (VFA) T1 mapping, the ordinary NLLS estimator aims to minimize the discrepancy between acquired signal data $\{y_n\}_{n=1}^N$ and model predictions $\{s_n(K, T1)\}_{n=1}^N$. This discrepancy is typically quantified as the sum of squared errors:

$$(\hat{K}, \hat{T1}) = \text{argmin}_{K,T1} \sum_{n=1}^{N} (y_n - s_n(K, T1))^2 \tag{3}$$

The above equation in terms of c_1 and c_2, can be rewritten as:

$$(\hat{K}, \hat{T1}) = \text{argmin}_{K,T1} \sum_{n=1}^{N} \left(y_n - \frac{c1 \sin \alpha_n}{1 - c2 \cos \alpha_n} \right)^2 \tag{4}$$

This optimization problem requires numerical algorithms to be solved. NOVIFAST is a novel NLLS optimization algorithm which combines the accuracy of NLLS estimators with the computational efficiency of heuristic linear algorithms. It uses the structure of SPGR model, and iteratively updates the estimate of the parameters K and $T1$ until convergence is achieved. Unlike descent-based algorithms, NOVIFAST does not require a descent property, simplifying the convergence criterion.
Additionally, NOVIFAST is designed to address problems in scenarios with noisy data. Signal-to-noise ratio (SNR) quantifies the ratio of the strength of a signal (acquired MRI data) to the level of background noise present in that signal. SNR is particularly relevant because it influences the performance and reliability of estimation algorithms. Higher SNR implies that the acquired signal is stronger relative to the background noise, making it easier to distinguish the signal from the noise, leading to more accurate parameter estimation. Lower SNR indicates that the signal is weaker compared to the noise, making it challenging for accurate estimation. In such scenarios, NOVIFAST is particularly useful and proves its robust parameter estimation even in the presence of noise.
SNR is given by the equation:

$$SNR = \frac{1}{N} \sum_{n=1}^{N} \frac{\sigma_n}{s_n}$$

The noise standard deviation σ is parameterized as $\sigma = \frac{K_{GT}}{SNR_{90}}$, with SNR_{90} representing the maximum SNR per pixel for an image acquired with an FA of 90 and $TR > 6 \times T1$.

2.3 Nelder-Mead Optimization

The Nelder-Mead algorithm, also known as the downhill simplex method, is a derivative-free optimization technique used to find the minimum (or maximum) of an objective function in a multi-dimensional space. It operates by iteratively transforming a simplex, which is a geometric figure consisting of $n + 1$ vertices in an n-dimensional space, towards the minimum of the objective function.

In the context of VFA T1 mapping, the Nelder-Mead algorithm is applied to optimize the parameters $c1$ and $c2$, and subsequently K and $T1$, by minimizing the sum of squared errors between the acquired signal data and the model predictions. By efficiently exploring the parameter space without relying on gradient information, Nelder-Mead optimization offers an alternative approach for accurate parameter estimation in the presence of noise and non-linearities.

2.4 Implicit Filtering Optimization

The implicit filtering algorithm operates by iteratively updating the current iterate based on the information provided by function evaluations at nearby points. Unlike traditional optimization methods that rely on derivatives or gradients of the objective function, implicit filtering only requires function values.

Implicit filtering does not explicitly build a model of the objective function but rather adapts the search direction based on local information obtained from function evaluations. This property makes it well-suited for optimization problems with noisy or expensive objective functions, as it can effectively navigate the search space without relying on gradient information.

In the context of VFA T1 mapping, implicit filtering can be applied to optimize the parameters (such as $c1$ and $c2$, and subsequently K and $T1$) by minimizing the discrepancy between the acquired signal data and the model predictions. By iteratively updating the parameter estimates based on function evaluations at nearby points, implicit filtering offers a robust and efficient approach for accurate parameter estimation in MRI relaxometry, particularly in scenarios with noisy data or complex objective functions.

In this study, we compare the performance of NOVIFAST, Nelder-Mead, and Implicit Filtering for VFA T1 mapping, evaluating their computational efficiency, and robustness to noise. By analyzing the strengths and weaknesses of these optimization techniques, we aim to provide insights into selecting the most suitable approach for T1 mapping in clinical settings.

3 Methodology

3.1 Data Acquisition

The dataset `"volume3DFSE.mat"` is a real 3D SPGR (Spoiled Gradient Recalled) MRI data, represented in a 4D format, where the fourth dimension corresponds to time. In MRI imaging, the three spatial dimensions (x, y, z) represent the spatial coordinates of the imaging volume, while the additional dimension (often referred to as the temporal dimension) represents the evolution of the imaging data over time. Therefore, the dimensions of the dataset allow for the exploration of both spatial and temporal variations in the MRI data. Each voxel in the data contains information about tissue properties such as relaxation times (T1), proton density, and spatial location within the imaging volume. The dataset provides a comprehensive representation of the anatomical structures and tissue characteristics captured during the MRI scanning process. The dataset provides a comprehensive representation of the anatomical structures and tissue characteristics captured during the MRI scanning process.

3.2 Novifast Method

Here's an overview of how the algorithm works:

1. **Objective function**: The objective function, given by equation (4) represents the sum of squared differences between the actual measured signal intensities (y_n) and the predicted signal intensities ($s_n(K, T1)$) for each voxel.

2. **Parameter Estimation**: The parameters, K and $T1$, are optimized to minimize the objective function.

3. **Input**: MRI Image data (could be either 3D or 4D, with multiple flip angles). In this project `volume3DFSE.mat` is used. The input is preprocessed to extract the size of image ('im'), number of slices, and the number of flip angles (α).

4. The algorithm computes the predicted signal intensities ($s_n(K, T1)$) based on the estimated parameters K and $T1$. These predicted signal intensities are computed for each voxel in the image.

5. The actual measured signal intensities (y_n) are extracted from the MRI image data for the regions identified by the mask (if provided). These measured signal intensities correspond to the regions of interest within the image.

6. The objective function is formulated as the sum of squared differences between the predicted and measured signal intensities for each voxel. This objective function is then minimized to estimate the parameters K and $T1$.

7. **Options**: Parameters for convergence control during optimization.

8. **Additional Parameters (varargin)**: Optional initial values and masks for the regions of interest. If not provided, it uses default values.

9. **Iterative Optimization**: The algorithm iteratively updates the estimates of c_1 and c_2, and consequently K and $T1$, until convergence criteria are met.

3.3 Nedler Mead Method

Here's an overview of how the Nelder-Mead algorithm works in case of this problem:

1. **Initial Simplex**: The algorithm begins with an initial simplex, which is initialized based on the initial parameter values `'ini'` provided.

2. **Reflection**: At each iteration, the algorithm evaluates the objective function, which measures the quality of the current solution (including parameter c2m) at the vertices of the simplex. It then performs a reflection operation to determine a new candidate point by reflecting across the centroid of the simplex from the worst vertex. This reflection operation corresponds to exploring a new candidate value for $c2m$ based on the current simplex configuration and evaluating its effect on the objective function.

3. **Expansion, Contraction, and Shrinkage**: Depending on the outcome of the reflection operation, the algorithm may perform additional operations such as expansion, contraction, or shrinkage to explore the objective function landscape efficiently. These operations correspond to adjusting the value of $c2m$ to explore different regions of the parameter space and evaluate their impact on the objective function.

4. **Update**: After evaluating the objective function at the new candidate point, the algorithm updates the simplex by replacing the worst vertex with the new point if it improves the objective function value.

5. **Convergence**: If the 'Direct' method is not specified in options structure ('options.Direct'), the algorithm iterates through these steps until a convergence criterion is met.

3.4 Implicit Filtering Optimization

Here's an overview of how the implicit filtering algorithm works:

1. **Initialization**: The algorithm starts with setting initial parameters K and $T1$ based on initial values `'ini'`.

2. **Objective Function Evaluation**: This step involves computing the objective function value (denoted as $f(x)$) and additional metrics (such as min_snm_ynm_values) required for optimization based on the current parameter values.

3. **Gradient Calculation**: Next, the gradient of the objective function is estimated using finite differences. This gradient (denoted as $\nabla f(x)$) provides information about the direction of steepest ascent.

4. **Backtracking Line Search**: The algorithm performs a backtracking line search to determine the step size (denoted as m) for updating the parameter values. This step ensures that the Armijo condition is satisfied.

5. **Update Parameters**: Based on the step size determined in the line search process, the parameter values (denoted as x) are updated to move towards the optimal solution. Specifically, the parameter $c2m$ is updated.

6. **Convergence Criterion**: If the 'Direct' method is not specified in options structure ('options.Direct'), the algorithm iterates through these steps until a convergence criterion is met.

4 Experiments

The Vivo T1 values of tissues were considered in the experiments to ensure accurate characterization and analysis of the MRI data.

4.1 Initial Setup

1. **Initial Guesses for T_1 and Proton Density (K):** The optimization process begins with initial guesses for T_1 and K. For all three algorithms, the initial guesses were set to [0.2, 500]. However, these values can be adjusted to better suit the specific imaging data and optimize the convergence process.

2. **Parameters:** The parameters to the function of the algorithms are passed in the 'options' structure. This structure allows for the customization of various aspects of the optimization process, including convergence criteria, maximum iterations,

and tolerance levels. Here are the key parameters utilized in the optimization algorithms:

(a) **MaxIter:** Maximum number of iterations.

(b) **Tol:** Tolerance defined for convergence criteria.

(c) **Direct:** Controls whether the optimization process includes a convergence criteria or not.

(d) **TR:** The repetition time, set as 9.

For implicit filtering, some more parameters are defined in addition to the above.

(a) **k:** A sequence used in the algorithm, employed in line search context.

(b) **c:** Armijo parameter, a constant used in the Armijo condition during line search.

(c) **rho:** Parameter within the range (0, 1) controlling the step size during line search.

(d) **amax:** Maximum value allowed for the backtracking parameter, preventing excessive iterations during line search.

3. **Convergence Criteria:** Both Nelder Mead and Implicit Filtering were run with and without convergence criteria. The convergence is controlled either by a maximum number of iterations or by a convergence criterion based on the l1 norm of $c2m$.

4.2 Code Execution

The file 'demo_novifast_image.m' is a demo script that utilizes the data 'volume3DFSE.mat' and calls the function 'novifast.m' to implement the NLLS algorithm. After the optimization, the computed T1 maps and convergence plot are visualized, and the computational time is printed. Similarly, the files, 'demo_nelder_mead_image.m' and 'demo_implicit_filtering_image.m' are demo scripts which call the respective algorithm functions.

4.3 Evaluation Metrics

The performance of each optimization algorithm was evaluated based on:

- **Computational Efficiency:** The computational efficiency of each algorithm was measured in terms of execution time. This metric provides insights into the speed of the algorithm to optimize the objective function.
- **Convergence Behavior:** The convergence behavior of each algorithm was analyzed using convergence plots. These plots depict the convergence of the objective function over iterations, providing visual feedback on the optimization progress and the algorithm's convergence characteristics.

5 Results

The results of the experimentation are summarized as follows:

5.1 With Convergence Criteria

The difference between predicted and actual signals for every iteration for all the methods are printed and these are the results:

1. NOVIFAST achieves a difference value accurate to 8.5899e-07 within 2.66 seconds and 27 iterations, before the convergence criterion based on the relative l1 norm for c2 is met, as shown in Figure 1.

2. Nelder Mead achieves a difference value of about 0.0163 in 0.4733 seconds within 10 iterations, where 10 is the maximum number of iterations specified, as shown in Figure 2.

3. Implicit Filtering achieves a difference value of about 0.0122 in 2.32 seconds within 27 iterations before the stopping criterion based on relative l1 norm for c2 is triggered.

The convergence plot are shown below:

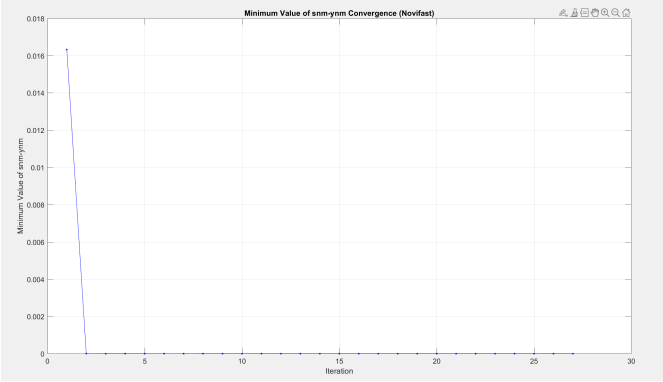

Figure 1: Convergence Plot of NOVIFAST

5.2 Without Convergence Criteria

When 'options.Direct' is specified, the difference between predicted and actual signals for every iteration for all the methods are printed and these are the results:

1. NOVIFAST achieves a difference value accurate to 3.7638e-06 in 0.286 seconds within 2 iterations, as shown in Figure 3, when ('Direct', 2) is given.

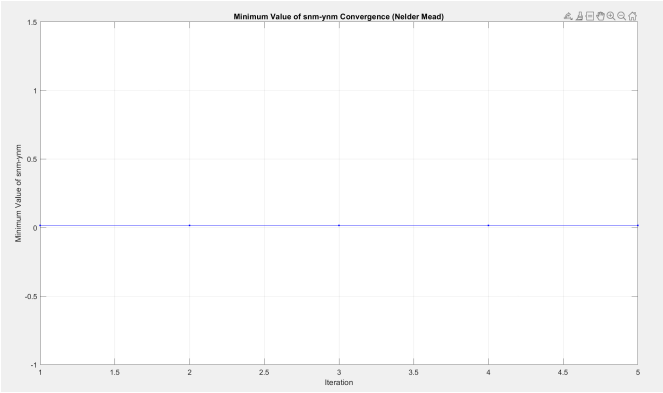

Figure 2: Convergence Plot of Nelder-Mead

2. Nelder Mead gives the difference value of about 0.0163 in 0.32 seconds within 10 iterations, as shown in Figure 4, where 10 is the maximum number of iterations specified.

3. Implicit Filtering gives the difference value of about 0.0121 in 2.23 seconds within 30 iterations, as shown in Figure 5, where 30 is the maximum number of iterations specified.

The convergence plot for all the three algorithms are plotted and shown below:

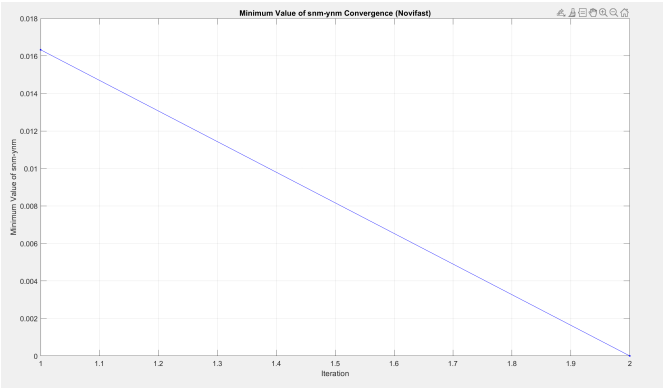

Figure 3: Convergence Plot of NOVIFAST (stopping-criteria free)

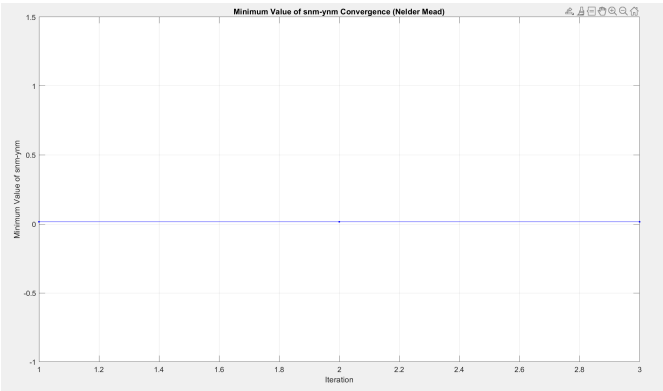

Figure 4: Convergence Plot of Nelder-Mead (stopping criteria free)

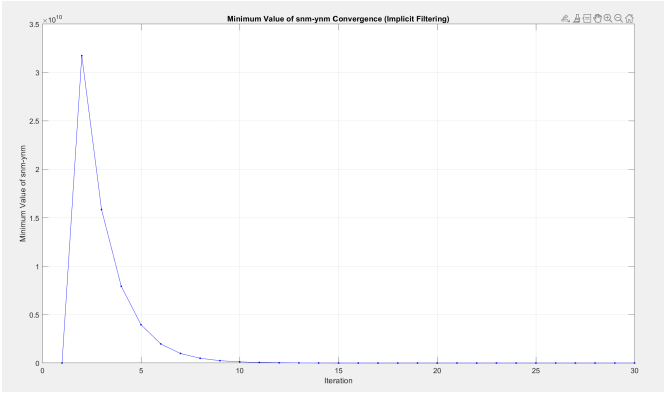

Figure 5: Convergence Plot of Implicit-Filtering (stopping criteria free)

A visualization of T1 map, displaying computational time for the three algorithms (without stopping criteria) is shown in Figure 6.

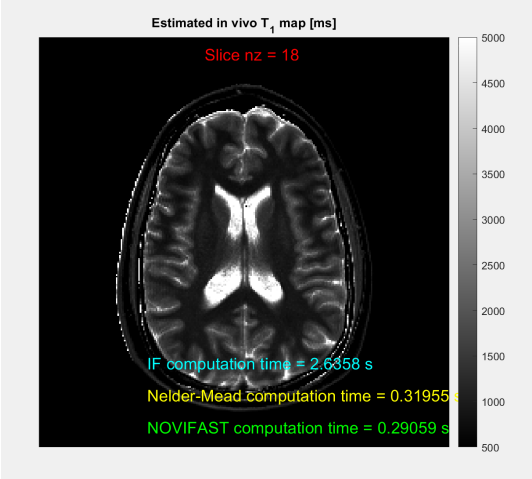

Figure 6: Visualization of T1 map

The performance of Novifast under different SNR levels and iterations are visualized in Figure 7, Figure 8, and Figure 9. Similar performance check can be done for Nelder Mead and Implicit Filtering in the future.

6 Discussion

It's evident that NOVIFAST stands out as the best method across various metrics such as computational efficiency, visualization of T1 maps, and convergence. Even in the presence of noise, NOVIFAST exhibits remarkable performance, making it a valuable tool in healthcare applications. This method, based on non-linear least squares, meticulously computes the function value at each iteration, showcasing its efficacy in medical imaging.
On the other hand, Nelder Mead method does not show good convergence. The fact that the minimum value of the difference between predicted and actual signals (snm-ynm) remains constant at approximately

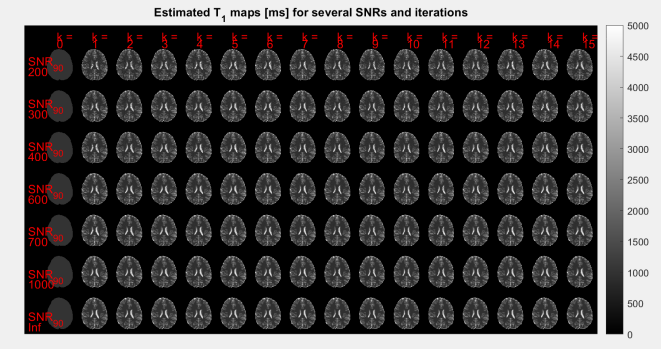

Figure 7: Estimated T_1 maps [ms] for several SNRs and iterations

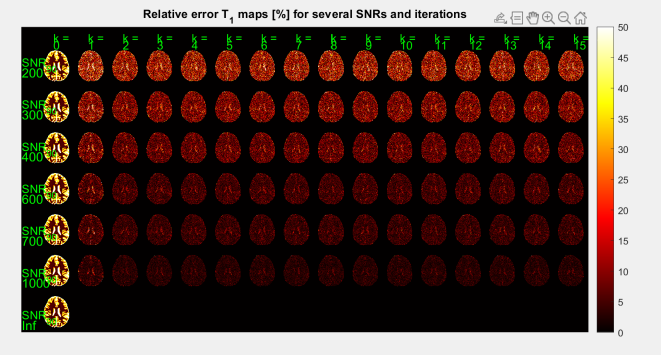

Figure 8: Relative error T_1 maps [%] for several SNRs and iterations

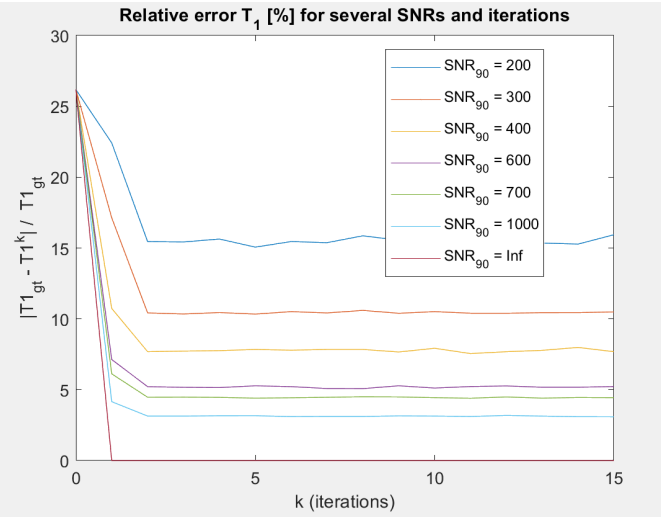

Figure 9: Relative error T_1 [%] for several SNRs and iterations

0.016327 throughout the iterations indicates that the algorithm may be stuck in a local minimum or struggling to converge towards the global minimum. The algorithm starts with a reflection step, where it evaluates the objective function at a point reflected through the centroid of the simplex. Then it encounters a high function value, hence it shrinks the simplex towards the best-performing point further. This iterative process of moving between reflection and shrinkage may result in an infinite loop, hence never converging to the global minima. Thus, it appears that the Nelder-Mead optimization method might not be well-suited for addressing this optimization problem.

Implicit filtering shows promising optimization performance, minimizing the objective function and converging towards parameter estimates. While the minimized value may not be as small as that achieved by NOVIFAST, this algorithm still converges within a reasonable number of iterations and showcases reliable convergence.

Implicit filtering outperforms Nelder Mead in this case due to its ability to efficiently handle optimization tasks with a large number of parameters. Unlike Nelder Mead, which relies on geometric operations such as reflection and contraction to update the simplex, implicit filtering employs a sequence of iterative steps based on implicit gradient information. Furthermore, implicit filtering's use of backtracking line search and adaptive step size adjustment allows it to adaptively adjust the step size during optimization, potentially leading to faster convergence and better handling of local minima or saddle points. This adaptive behavior enables implicit filtering to explore the optimization landscape more effectively.

7 Future Scope

In addition to comparing NOVIFAST, Nelder Mead, and implicit filtering, further work can be done:

1. **Robustness to Noise:** Evaluate the optimization algorithms based on their robustness to noise. While NOVIFAST exhibits remarkable performance, assessing its performance against increasing levels of noise compared to Nelder Mead and implicit filtering can provide insights into their relative strengths.

2. **Parameter Tuning:** Investigate the impact of parameter tuning on the methods. Identify specific parameters that significantly influence performance and assess the ease of effectively tuning these parameters for users.

8 References

References

[1] Maurovich-Horvat, P., Ferencik, M., Voros, S., Merkely, B., & Hoffmann, U. (2018). Comprehensive plaque assessment by coronary CT angiography. *Nature Reviews Cardiology*, 15(6), 303-313. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6277233/

[2] Wilm, B. J., Barmet, C., Pruessmann, K. P., & Kasper, L. (2018). Novifast: A Fast Algorithm for Accurate and Precise VFA MRI. Retrieved from https://www.mathworks.com/matlabcentral/fileexchange/67815

[3] Author(s). (Year). Title of the article. *Journal Name*, Volume(Issue), Page numbers. Retrieved from https://pubmed.ncbi.nlm.nih.gov/22807160

[4] Author(s). (Year). Title of the article. Retrieved from https://www.math.uci.edu/~qnie/Publications/NumericalOptimization.pdf

1 Appendix

'demo_novifast_image.m'

```
clear all; clc; close all;

addpath('./data')
load('volume3DFSE.mat') % 3D FSE SPGR dataset (1.5 T) used in in vivo experiment of '....'

% NOVIFAST's parameter definition
TR = 9; % Repetition time [ms]
ini = [0.2, 500]; % K [] and T1 [ms] initial constant maps for NOVIFAST
options = struct('Direct', 2); % If field 'Direct' is given, it means NOVIFAST is run in a blind mod
options.MaxIter = 100;
options.TR = 9;

% Call NOVIFAST
time = tic;
[K, T1, min_snm_ynm_values_novifast] = novifast_image(im, alpha, TR, options);
time = toc(time);

% Display final computational time
disp(['Dataset computation time = ', num2str(time), ' s']);

% Plot the minimum values of snm-ynm against the iteration number for NOVIFAST
figure;
plot(1:length(min_snm_ynm_values_novifast), min_snm_ynm_values_novifast, 'b.-');
xlabel('Iteration');
ylabel('Minimum Value of snm-ynm');
title('Minimum Value of snm-ynm Convergence (Novifast)');
grid on;
```

'novifast_image.m'

```
function [ K, T1, min_snm_ynm_values ] = novifast_image( Im, alpha, TR, options , varargin )
    %% check input
    sizeim=size(Im);
    disp(['Size of Im: ', num2str(sizeim)]);
    nrows=sizeim(1);
    ncols=sizeim(2);

    % Check if the input image is 3D or 4D
    if numel(sizeim)==3
        nslices=1;
        nalpha=sizeim(3);
    else
        nslices=sizeim(3);
        nalpha=sizeim(4);
    end

    % Check FA
    N=numel(alpha);

    if nalpha~=N
        error('Dimensions do not match');
    end

    if isempty(nslices)
        nslices=1;
```

```
end

disp(options);
% Checks the options structure passed to the function and sets the convergence parameters accora
if ~isfield(options, 'Direct')          % Checks if the field ''Direct'' is not present in the o
    if ~isfield(options, 'MaxIter')
        options.MaxIter = 10; %default
    elseif options.MaxIter<1 || options.MaxIter>200
        error('options:-Maxiter-should-be-set-to-a-value-between-1-and-200');
    end

    if ~isfield(options,'Tol')
        options.Tol = 1e-6; %default
    elseif options.Tol<0 || options.Tol>1e-2
        error('options:-Tol-should-be-set-to-a-value-between-0-and-1e-2');
    end
    modeDirect=false;       % modeDirect to false. This indicates that the convergence mode is n
elseif options.Direct<1 || options.Direct>200
    error('options:-Directiter-should-be-set-to-a-value-between-1-and-200');
else
    modeDirect=true;
end

if isempty(varargin)
    fprintf('User-did-not-provide-initial-values-neither-a-mask.-Using-default-parameters...\n')
    ini=[0.5,500];
    th=0.05*max(max(max(Im(:))));  %Intensity values smaller than 5% of the maximum value of the
    if nslices==1
        mask = squeeze(Im(:,:,1))>th;
    else
        mask = squeeze(Im(:,:,:,1))>th;
    end
elseif ~isvector(varargin{1})
     fprintf('User-did-not-provide-initial-values.-Using-default-parameters...-\n');
     ini=[0.5,500];
     mask=varargin{1};
else
    ini=varargin{1};
    if length(varargin)==1
        fprintf('User-did-not-provide-a-mask.-Using-default-parameters...-\n');
        th=0.05*max(max(max(Im(:))));  %Intensity values smaller than 5% of the maximum value of
        if nslices==1
            mask = squeeze(Im(:,:,1))>th;
        else
            mask = squeeze(Im(:,:,:,1))>th;
        end
    else
        fprintf('User-did-provide-initial-values-and-a-mask-\n')
        mask=varargin{2};
    end
end

pm=find(mask);
M=numel(pm);
%% NOVIFAST begins here

% pre-computing
K=squeeze(zeros(nrows,ncols,nslices));
T1=squeeze(zeros(nrows,ncols,nslices));
```

```
% Initialize min_snm_ynm_values array to store minimum values of snm-ynm
min_snm_ynm_values = [];

alphanm=alpha*ones(1,M);
y=reshape(Im(:),nrows*ncols*nslices,N);
ynm=y(pm,:)';
pnm=sind(alphanm);
qnm=cosd(alphanm);

% initialization
Kini=ini(1);
T1ini=ini(2);
c1m=Kini*(1*exp(-TR/T1ini))*ones(1,M);
c2m=exp(-TR/T1ini)*ones(1,M);
k=0;
done=false;


% iterative process
while ~done

    c2m_old=c2m;
    c1m=repmat(c1m,[N,1]);
    c2m=repmat(c2m,[N,1]);
    denm=1-c2m.*qnm;
    snm=c1m.*pnm./denm;

    %definition of vectors
    A=ynm.*qnm./denm;
    Ahat=snm.*qnm./denm;
    B=pnm./denm;
    Z=ynm./denm;


    %definition of inner products
    BB=sum(B.^2,1);
    AAhat=sum(A.*Ahat,1);
    BAhat=sum(B.*Ahat,1);
    BA=sum(B.*A,1);
    BZ=sum(B.*Z,1);
    ZAhat=sum(Z.*Ahat,1);

    %calculation of c1m and c2m
    detm=BB.*AAhat- BAhat.*BA;
    c1m=(BZ.*AAhat - ZAhat.*BA)./detm;
    c2m=(BB.*ZAhat - BAhat.*BZ)./detm;
    k=k+1;

    min_snm_ynm = min(abs(snm - ynm), [], 'all');
    min_snm_ynm_values = [min_snm_ynm_values, min_snm_ynm];

    disp(['Iteration ', num2str(k), ': Minimum value of snm-ynm = ', num2str(min_snm_ynm)]);

    %stopping
    if modeDirect %mode with no-convergence criterion
        if k==options.Direct
            done=true;
        end
    else
        rel_err=( norm(c2m(isfinite(c2m))-c2m_old(isfinite(c2m_old)),1) )/ norm(c2m(isfinite(c2m
```

```
%Relative l1 norm for c2 (Convergence is controlled by c2 only)
            if rel_err< options.Tol || k>=options.MaxIter %mode with convergence criterion
                fprintf("Convergence Criteria Triggered.")
                done=true;
            end
        end
    end

    Km=c1m./(1-c2m);
    T1m=-TR./log(c2m);

    %K and T1 maps
    T1(pm)=T1m;
    K(pm)=Km;

end
```

'demo_nelder_mead_image.m'

```
%clear all; clc; close all;

addpath('./data')
load('volume3DFSE.mat') % 3D FSE SPGR dataset (1.5 T) used in in vivo experiment of '....'

% Nelder Mead's parameter definition
TR = 9; % Repetition time [ms]
ini = [0.2, 500]; % K [] and T1 [ms] initial constant maps
options = struct('Direct', 2); % If field 'Direct' is given, it means the code is run in a blind mod
options.Tol = 0;
options.MaxIter = 5;

% Call Nelder Mead
time = tic;
[K, T1, min_snm_ynm_values_nm] = nelder_mead_optimization(im, alpha, TR, options);
time = toc(time);

% Display final computational time
disp(['Dataset computation time = ', num2str(time), ' s']);

% Plot the minimum values of snm-ynm against the iteration number for NOVIFAST
figure;
plot(1:length(min_snm_ynm_values_nm), min_snm_ynm_values_nm, 'b.-');
xlabel('Iteration');
ylabel('Minimum Value of snm-ynm');
title('Minimum Value of snm-ynm Convergence (Nelder Mead)');
grid on;
```

'nelder_mead_optimization.m'

```
function [ K, T1, min_snm_ynm_values ] = nelder_mead_optimization( Im, alpha, TR, options, varargin

    %% Check input
    sizeim = size(Im);
    disp(['Size of Im: ', num2str(sizeim)]);
    nrows = sizeim(1);
    ncols = sizeim(2);

    % Check if the input image is 3D or 4D
```

```
if numel(sizeim) == 3
    nslices = 1;
    nalpha = sizeim(3);
else
    nslices = sizeim(3);
    nalpha = sizeim(4);
end

% Check FA
N = numel(alpha);

if nalpha ~= N
    error('Dimensions do not match');
end

if isempty(nslices)
    nslices = 1;
end

% Checks the options structure passed to the function and sets the convergence parameters accord
if ~isfield(options, 'Direct') % Checks if the field ''Direct'' is not present in the options st
    if ~isfield(options, 'MaxIter')
        options.MaxIter = 10; % Default
    elseif options.MaxIter < 1 || options.MaxIter > 200
        error('options: Maxiter should be set to a value between 1 and 200');
    end

    if ~isfield(options, 'Tol')
        options.Tol = 1e-6; % Default
    elseif options.Tol < 0 || options.Tol > 1e-2
        error('options: Tol should be set to a value between 0 and 1e-2');
    end
    modeDirect = false; % Indicates that the convergence mode is not set to direct.
elseif options.Direct < 1 || options.Direct > 200
    error('options: Directiter should be set to a value between 1 and 200');
else
    modeDirect = true;
end

if isempty(varargin)
    fprintf('User did not provide initial values neither a mask. Using default parameters...\n')
    ini = [0.5, 500];
    th = 0.05 * max(max(max(Im(:)))); % Intensity values smaller than 5% of the maximum value of
    if nslices == 1
        mask = squeeze(Im(:,:,1)) > th;
    else
        mask = squeeze(Im(:,:,:,1)) > th;
    end
elseif ~isvector(varargin{1})
    fprintf('User did not provide initial values. Using default parameters... \n');
    ini = [0.5, 500];
    mask = varargin{1};
else
    ini = varargin{1};
    if length(varargin) == 1
        fprintf('User did not provide a mask. Using default parameters... \n');
        th = 0.05 * max(max(max(Im(:)))); % Intensity values smaller than 5% of the maximum valu
        if nslices == 1
            mask = squeeze(Im(:,:,1)) > th;
        else
```

```
            mask = squeeze(Im(:,:,:,1)) > th;
        end
    else
        fprintf('User did provide initial values and a mask\n')
        mask = varargin{2};
    end
end

pm = find(mask);
M = numel(pm);

%% Nelder Mead begins here

% Pre-computing
K = zeros(nrows, ncols, nslices); % Initialize K with the specified size
T1 = zeros(nrows, ncols, nslices); % Initialize T1 with the specified size

alphanm = alpha * ones(1, M);
y = reshape(Im(:), nrows * ncols * nslices, N);
ynm = y(pm, :)';
pnm = sind(alphanm);
qnm = cosd(alphanm);

% Initialization
Kini = ini(1);
T1ini = ini(2);
c1m = Kini * (1 * exp(-TR/T1ini)) * ones(1, M);
c2m = exp(-TR/T1ini) * ones(1, M);

% Initialize min_snm_ynm_values array to store minimum values of snm-ynm
min_snm_ynm_values = [];

% Nelder mead step instead of iterative process
[K, T1, min_snm_ynm_values] = nelder_mead_step(K, T1, c1m, c2m, ynm, pnm, qnm, alpha, N, TR, opt

end

function [ K, T1, min_snm_ynm_values ] = nelder_mead_step(K, T1, c1m, c2m, ynm, pnm, qnm, alpha, N,

pm = find(mask);
M = numel(pm);
% Initialize the simplex with the provided c1m and c2m
simplex = [c1m(:), c2m(:)];
% Number of variables (dimension)
n = size(simplex, 2);
% Number of vertices (including the best and worst points)
m = n + 1;

% Initialize objective function values for the simplex
f_values = zeros(m, 1);
for i = 1:m
    % Evaluate the objective function for each vertex of the simplex
    f_values(i) = objective_function(simplex(i, :), c1m, c2m, ynm, pnm, qnm, alpha, TR, M);
end

% Main loop of the Nelder-Mead algorithm
iter = 0;
done = false;
min_snm_ynm_values = [];
```

```
while iter < options.MaxIter && ~done % Termination based on maximum number of iterations
    % Update c2m_old at the beginning of each iteration
    c2m_old = c2m;

    % Sort the vertices of the simplex based on the objective function values
    [f_values, order] = sort(f_values);
    simplex = simplex(order, :);

    % Compute the centroid of all vertices except the worst one
    centroid = mean(simplex(1:end-1, :));

    % Reflection: Compute the reflected point and evaluate the objective function
    x_reflected = centroid + (centroid - simplex(end, :));
    f_reflected = objective_function(x_reflected, c1m, c2m, ynm, pnm, qnm, alpha, TR, M);
    % Reflection
    disp('Reflection');
    disp(['f_reflected:-', num2str(f_reflected)]);

    if f_reflected < f_values(end-1) && f_reflected >= f_values(1)
        % Replace the worst point with the reflected point
        simplex(end, :) = x_reflected;
        f_values(end) = f_reflected;
        % Update c2m with the second element of the reflected point
        c2m = x_reflected(:, 2);
    elseif f_reflected < f_values(1)
        % Expansion
        disp('Expansion');
        x_expanded = centroid + 2 * (x_reflected - centroid);
        f_expanded = objective_function(x_expanded, c1m, c2m, ynm, pnm, qnm, alpha, TR, M);
        if f_expanded < f_reflected
            % Replace the worst point with the expanded point
            simplex(end, :) = x_expanded;
            f_values(end) = f_expanded;
            % Update c2m with the second element of the expanded point
            c2m = x_expanded(:, 2);
        else
            % Replace the worst point with the reflected point
            simplex(end, :) = x_reflected;
            f_values(end) = f_reflected;
            % Update c2m with the second element of the reflected point
            c2m = x_reflected(:, 2);
        end
    else
        % Contraction
        disp('Contraction');
        x_contracted = centroid + 0.5 * (simplex(end, :) - centroid);
        f_contracted = objective_function(x_contracted, c1m, c2m, ynm, pnm, qnm, alpha, TR, M);

        if f_contracted < f_values(end)
            % Replace the worst point with the contracted point
            simplex(end, :) = x_contracted;
            f_values(end) = f_contracted;
            % Update c2m with the second element of the contracted point
            c2m = x_contracted(:, 2);
        else
            % Shrink the simplex towards the best point
            disp('Shrink');
            for i = 2:m
                % Shrink the simplex
                simplex(i, :) = 0.5 * (simplex(i, :) + simplex(1, :));
```

```
            end
        end
        % Update min_snm_ynm_values with the minimum value of snm-ynm
        min_snm_ynm = min(abs(c1m .* pnm ./ (1 - c2m_old .* qnm) - ynm), [], 'all');
        min_snm_ynm_values = [min_snm_ynm_values, min_snm_ynm];
    end


    K(pm) = simplex(1, 1);
    T1(pm) = -TR / log(simplex(1, 2));



    % Check convergence based on stopping criteria
    if modeDirect
        % Mode with no-convergence criterion
        if iter == options.Direct
            done = true;
        end
    else
        % Mode with convergence criterion
        rel_err = (norm(c2m(isfinite(c2m)) - c2m_old(isfinite(c2m_old)), 1)) / norm(c2m(isfinite
        fprintf('rel_err:%f\n', rel_err);

        if rel_err < options.Tol || iter >= options.MaxIter
            fprintf('Convergence criteria triggered.');
            done = true;
        end
    end


    iter = iter + 1; % Increment iteration count
    disp(['Minimum value of snm-ynm =', num2str(min_snm_ynm_values(end))]);
end

% Return the best solution found
K = K;
T1 = T1;
end


function f_value = objective_function(x, c1m, c2m, ynm, pnm, qnm, alpha, TR, M)

    % Evaluate the objective function (sum of squared differences)

    % Extract parameters from x
    N = numel(alpha);

    % Compute snm based on the current parameters
    denm = 1 - c2m .* qnm;
    snm = c1m .* pnm ./ denm;

    % Compute the sum of squared differences between ynm and snm
    squared_diff = sum((ynm - snm).^2, 1);

    % Compute the objective function value
    f_value = sum(squared_diff);
end
```

'demo_implicit_filtering_image.m'

```
%clear all; clc; close all;

addpath('./data')
load('volume3DFSE.mat') % 3D FSE SPGR dataset (1.5 T) used in in vivo experiment of '....'

% Implicit Filtering's parameter definition
TR = 9; % Repetition time [ms]
ini = [0.2, 500]; % K [] and T1 [ms] initial constant maps

% Options structure
%options_imfi = struct('Direct', 1);
options_imfi.k = @(k) 1 / (2^k); % Example of defining the sequence k
options_imfi.c = 0.5; % Armijo parameter
options_imfi.rho = 0.5; % Parameter in (0, 1)
options_imfi.amax = 100; % Maximum backtracking parameter
options_imfi.Tol = 1e-6; % Convergence tolerance
options_imfi.MaxIter = 30; % Maximum number of iterations

% Call Implicit Filtering
time = tic;
[K, T1, min_snm_ynm_values_imfi] = implicit_filtering_optimization(im, alpha, TR, options_imfi);
time = toc(time);

% Display final computational time
disp(['Dataset computation time = ', num2str(time), ' s']);

% Plot the minimum values of snm–ynm against the iteration number for NOVIFAST
figure;
plot(1:length(min_snm_ynm_values_imfi), min_snm_ynm_values_imfi, 'b.-');
xlabel('Iteration');
ylabel('Minimum Value of snm–ynm');
title('Minimum Value of snm–ynm Convergence (Implicit Filtering)');
grid on;
```

'implicit_filtering_optimization.m'

```
function [K, T1, min_snm_ynm_values] = implicit_filtering_optimization(Im, alpha, TR, options_imfi,
    %% check input
    sizeim=size(Im);
    disp(['Size of Im: ', num2str(sizeim)]);
    nrows=sizeim(1);
    ncols=sizeim(2);

    % Check if the input image is 3D or 4D
    if numel(sizeim)==3
        nslices=1;
        nalpha=sizeim(3);
    else
        nslices=sizeim(3);
        nalpha=sizeim(4);
    end

    % Check FA
    N=numel(alpha);

    if nalpha~=N
        error('Dimensions do not match');
```

```
end

if isempty(nslices)
    nslices=1;
end

disp(options_imfi);  % Print the contents of options_imfi to inspect its structure

% Checks the options structure passed to the function and sets the convergence parameters accord
if ~isfield(options_imfi, 'Direct')          % Checks if the field ''Direct'' is not present in
    if ~isfield(options_imfi, 'MaxIter')
        options_imfi.MaxIter = 10; %default
        modeDirect=false
    elseif options_imfi.MaxIter<1 || options_imfi.MaxIter>200
        error('options:-Maxiter-should-be-set-to-a-value-between-1-and-200');
    end

    if ~isfield(options_imfi,'Tol')
        options_imfi.Tol = 1e-6; %default
    elseif options_imfi.Tol<0 || options_imfi.Tol>1e-2
        error('options:-Tol-should-be-set-to-a-value-between-0-and-1e-2');
    end
    modeDirect=false;        % modeDirect to false. This indicates that the convergence mode is n
elseif options_imfi.Direct<1 || options_imfi.Direct>200
    error('options:-Directiter-should-be-set-to-a-value-between-1-and-200');
else
    modeDirect=true;
end

if isempty(varargin)
    fprintf('User-did-not-provide-initial-values-neither-a-mask.-Using-default-parameters...\n')
    ini=[0.2,500];
    th=0.05*max(max(max(Im(:)))); %Intensity values smaller than 5% of the maximum value of the
    if nslices==1
        mask = squeeze(Im(:,:,1))>th;
    else
        mask = squeeze(Im(:,:,:,1))>th;
    end
elseif ~isvector(varargin{1})
     fprintf('User-did-not-provide-initial-values.-Using-default-parameters...-\n');
     ini=[0.5,500];
     mask=varargin{1};
else
    ini=varargin{1};
    if length(varargin)==1
        fprintf('User-did-not-provide-a-mask.-Using-default-parameters...-\n');
        th=0.05*max(max(max(Im(:)))); %Intensity values smaller than 5% of the maximum value of
        if nslices==1
            mask = squeeze(Im(:,:,1))>th;
        else
            mask = squeeze(Im(:,:,:,1))>th;
        end
    else
        fprintf('User-did-provide-initial-values-and-a-mask-\n')
        mask=varargin{2};
    end
end

pm=find(mask);
M=numel(pm);
```

```
% Pre-computing
K = zeros(nrows, ncols, nslices);
T1 = zeros(nrows, ncols, nslices);
alphanm = alpha * ones(1, M);
y = reshape(Im(:), nrows * ncols * nslices, N);
ynm = y(pm, :)';
pnm = sind(alphanm);
qnm = cosd(alphanm);

% Initialization

x = ini;
k = 1;
%increment_k = false;
% initialization
Kini=ini(1);
T1ini=ini(2);
c1m\_old=Kini*(1*exp(-TR/T1ini))*ones(1,M);
c2m_old=exp(-TR/T1ini)*ones(1,M);
done = false;
iter = 0;

% Initialize min_snm_ynm_values array to store minimum values of snm-ynm
min_snm_ynm_values = [];

% Implicit Filtering loop

while iter < options_imfi.MaxIter && ~done

% Compute f(x) and  f (x)
%f_x = objective_function(x, alpha, M, ynm);
[f_x, min_snm_ynm_values] = objective_function(x, alpha, M, TR, ynm, min_snm_ynm_values, iter);

%grad_f_x = finite_difference_gradient(x, alpha, M, ynm);
grad_f_x = finite_difference_gradient(x, alpha, M, TR, ynm, min_snm_ynm_values, iter);

% Check Armijo condition
if norm(grad_f_x) <= options_imfi.k(k)
    done = true;
else

    % Find smallest integer m between 0 and amax
    m = 0;
    x_temp = x;
    f_x_temp = f_x;

    while m < options_imfi.amax
        % Compute new x and f(x) using backtracking line search

        x_temp = x - options_imfi.rho^m * grad_f_x;
        %[f_x_temp, min_snm_ynm_values, c1, c2] = objective_function(x_temp, alpha, M, ynm, min_
        [f_x_temp, min_snm_ynm_values, c2m, c2m_old] = objective_function(x_temp, alpha, M, TR,
        if f_x_temp <= f_x - options_imfi.c * options_imfi.rho^m * (grad_f_x' * grad_f_x)
            break;
        end
        m = m + 1;

    end
    if m == options_imfi.amax
```

```
            done = true;
        else
            x = x_temp;
        end
    end

    % Update iteration count
    iter = iter + 1;

    % Print minimum value of snm-ynm for each iteration
    disp(['Minimum value of snm-ynm = ', num2str(min_snm_ynm_values(end))]);
    %stopping
    %min_snm_ynm = min(abs(c1m .* pnm ./ (1 - c2m .* qnm) - ynm), [], 'all');
    %min_snm_ynm_values = [min_snm_ynm_values, min_snm_ynm];

    if modeDirect %mode with no-convergence criterion
        if isfield(options_imfi, 'Direct') && k == options_imfi.Direct
            done = true;
        end
    else
        rel_err = (norm(c2m(isfinite(c2m)) - c2m_old(isfinite(c2m_old)), 1)) / norm(c2m(isfinite(c2m
        fprintf('rel_err: %f, k: %d\n', rel_err, k);

        if rel_err < options_imfi.Tol || k >= options_imfi.MaxIter
            fprintf('Convergence criteria triggered. rel_err: %f, k: %d\n', rel_err, k);
            done = true;
        end

    end


    % Update K and T1
    K(pm) = x(1);
    T1(pm) = x(2);


    % Update min_snm_ynm_values with the minimum value of snm-ynm
    %min_snm_ynm = min(abs(c1m .* pnm ./ (1 - c2m .* qnm) - ynm), [], 'all');
    %min_snm_ynm_values = [min_snm_ynm_values, min_snm_ynm];


    end

    % Check the size of T1
    disp(['Size of T1: ', num2str(size(T1))]);
    % Print statements moved outside the loop
    %disp(['Objective Function - Iteration ', num2str(iter), ': Minimum value of snm-ynm = ', num2st
    %disp(['Finite Difference Gradient - Iteration ', num2str(iter), ': Calculating gradient for par
end

function [f_value, min_snm_ynm_values, c2m, c2m_old] = objective_function(x, alpha, M, TR, ynm, min_
    % Evaluate the objective function (sum of squared differences)
    % Extract parameters from x

    %c1 = x(1);
    %c2 = x(2);
    N=numel(alpha);

    Kini = x(1);
    T1ini = x(2);
```

```
    c1m=Kini*(1*exp(-TR/T1ini))*ones(1,M);
    c2m=exp(-TR/T1ini)*ones(1,M);
    c2m_old=c2m;

    c1m=repmat(c1m,[N,1]);
    c2m=repmat(c2m,[N,1]);

    alphanm=alpha*ones(1,M);
    pnm=sind(alphanm);
    qnm=cosd(alphanm);

    % Compute snm based on the current parameters
    denm = 1 - c2m .* qnm;
    snm = c1m.* pnm ./ denm;

    % Compute the sum of squared differences between ynm and snm
    squared_diff = sum((ynm - snm).^2, 1);

    % Compute minimum value of snm-ynm
    min_snm_ynm = min(abs(c1m .* pnm ./ (1 - c2m .* qnm) - ynm), [], 'all');

    % Store minimum value in the list
    min_snm_ynm_values = [min_snm_ynm_values, min_snm_ynm];

    %disp(['Iteration ', num2str(iter), ': Minimum value of snm-ynm = ', num2str(min_snm_ynm)]);
    %disp(['Objective Function - Iteration ', num2str(iter), ': Minimum value of snm-ynm = ', num2st

    % Compute the objective function value
    f_value = sum(squared_diff);
end

function grad = finite_difference_gradient(x, alpha, M, TR, ynm, min_snm_ynm_values, iter)
    % Compute gradient using finite differences
    grad = zeros(size(x));
    h = 1e-6; % Step size for finite differences

    %for i = 1:numel(x)
    %    x_plus_h = x;
    %    x_plus_h(i) = x_plus_h(i) + h;
    %    grad(i) = (objective_function(x_plus_h, alpha, M, ynm, min_snm_ynm_values, iter) - objectiv
    %end

    for i = 1:numel(x)
        x_plus_h = x;
        x_plus_h(i) = x_plus_h(i) + h;

        obj_func_before = objective_function(x, alpha, M, TR, ynm, min_snm_ynm_values, iter);
        obj_func_after = objective_function(x_plus_h, alpha, M, TR, ynm, min_snm_ynm_values, iter);

        grad(i) = (obj_func_after - obj_func_before) / h;

        %disp(['Finite Difference Gradient - Iteration ', num2str(iter), ': Calculating gradient for
    end
end
```

'visualize_T1_Map.m'

```
clear all; clc; close all;
```

```
addpath('./data')
load('volume3DFSE.mat') % 3D FSE SPGR dataset (1.5 T) used in in vivo experiment

% NOVIFAST's parameter definition
TR = 9; % Repetition time [ms]
ini = [0.2, 500]; % K [] and T1 [ms] initial constant maps for NOVIFAST
options_novifast = struct('Direct', 2); % If field 'Direct' is given, it means NOVIFAST is run in a

% Call NOVIFAST and measure time
time_novifast = tic;
[K_novifast, T1_novifast] = novifast_image(im, alpha, TR, options_novifast);
time_novifast = toc(time_novifast);

% Nelder-Mead's parameter definition
options_nelder = struct('Direct', 2); % Run in a blind mode, i.e., no convergence criterion.
options_nelder.MaxIter = 100;

% Call Nelder-Mead and measure time
time_nelder = tic;
[K_nelder, T1_nelder] = nelder_mead_optimization(im, alpha, TR, options_nelder);
time_nelder = toc(time_nelder);

% IF's parameter definition
%options_if = struct('Direct', 2); % Run in a blind mode, i.e., no convergence criterion.
options_if=struct('Direct',2); %If field 'Direct' is given, it means NOVIFAST is run in a blind mode
options_if.step_size = 0.01; % Set the step size value according to your needs
options_if.max_backtracking = 20; % Set the maximum number of backtracking steps
options_if.rho = 0.5;
options_if.c = 0.5; % Set the Armijo condition parameter (c) value
options_if.MaxIter = 100;

options_imfi = struct('Direct', 1);
options_imfi.k = @(k) 1 / (2^k); % Example of defining the sequence k
options_imfi.c = 0.5; % Armijo parameter
options_imfi.rho = 0.5; % Parameter in (0, 1)
options_imfi.amax = 100; % Maximum backtracking parameter
options_imfi.Tol = 1e-6; % Convergence tolerance
options_imfi.MaxIter = 100; % Maximum number of iterations

% Call IF and measure time
time_if = tic;
[K_if, T1_if] = implicit_filtering_optimization(im, alpha, TR, options_imfi);
time_if = toc(time_if);


% Display computation times
disp(['Computation time for NOVIFAST: ', num2str(time_novifast), ' seconds']);
disp(['Computation time for Nelder-Mead: ', num2str(time_nelder), ' seconds']);
disp(['Computation time for Implicit-filtering: ', num2str(time_if), ' seconds']);
%disp(['Computation time for Implicit-filtering: ', num2str(time_if), ' seconds']);

% Visualization
warning off
figure(1)
if numel(size(im)) == 4 % If im is a volume, visualize middle slice
nslice = 18;
imshow(squeeze(T1_novifast(:,:,nslice)), [500, 5000]) % A single slice is visualized
title('Estimated in vivo T_1 map [ms]')
colorbar
elseif numel(size(im)) == 3
```

```
nslice = 1;
imshow(T1_novifast, [500, 5000])
title('Estimated in vivo T_1 map [ms]')
colorbar
end
strg = ['Slice nz=', num2str(nslice)];
text(104, 11, strg, 'fontsize', 14, 'Color', 'red')
strg2 = ['NOVIFAST computation time=', num2str(time_novifast), 's'];
text(68, 244, strg2, 'fontsize', 14, 'Color', 'green')
% Add computation time for Nelder–Mead to the figure
strg3 = ['Nelder–Mead computation time=', num2str(time_nelder), 's'];
text(68, 224, strg3, 'fontsize', 14, 'Color', 'yellow')
% Add computation time for IF to the figure
strg4 = ['IF computation time=', num2str(time_if), 's'];
text(68, 204, strg4, 'fontsize', 14, 'Color', 'cyan')
```