Herbert Heigl

CUDA powered MRI reconstruction with variational constraints

Master Thesis

Graz University of Technology

Institute of Medical Engineering
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Stollberger Rudolf

Supervisor: Univ.-Prof. Dipl-Ing.,Dr.techn. Stollberger Rudolf

Graz, January 2019

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

The acceleration of magnetic resonance imaging (MRI) has been a central research topic for many years.
In addition to the correct measurement of time-dependent processes and the reduction of motion artifacts, the acceleration of MRT is of particular importance for the clinical applicability of new specific examination methods.
Recently, work has focused in particular on subsampled ”Parallel Imaging” methods. Only subsegments of the data necessary for a conventional reconstruction are acquired, but these are measured with several receiving coils in parallel. In combination with new mathematical methods it is possible to reconstruct artifact-free images or scans with high temporally resolution from these accelerated, mutli-coil measurements.
With a special technique, the iteratively regularized Gauss-Newton (IRGN) method, it is also possible to determine the influence of the spatially variable receiver coil sensitivity in the reconstruction.
The runtime-optimal implementation of this sophisticated reconstruction procedure in combination with different regularization techniques (Tikhonov L2-norm, ”total variation” and ”total generalized variation”) is the subject of this work.
To achieve fast reconstruction we leverage the power of high-end GPUs with CUDA and compared the results to a reference implementation in Matlab. In this thesis the mathematical background is described and then implementation techniques for a fast image reconstruction are discussed.
A special challenge of the inherent nonlinear inverse problem is the Primal-Dual Extra-Gradient Algorithm for IRGN, which causes a high computational load. In order to accelerate the matrix-based calculations, the existing C++ Agile Library of the Institute of Medical Engineering was extended to include a CUDA framework for GPU image reconstruction.
The GPU implementation made it possible to reduce the image reconstruction time to one tenth of the reference implementation.
This work was concluded with a comparative analysis of the reconstruction

quality by graphics card compared to a reference implementation in Matlab. The visual impression of the tissue signals showed no immediately recognizable differences. In the subtraction analysis differences occurred primarily at the tissue edges. The results of the numerical evaluation showed deviations which are below the typical noise level.

Keywords: Magnetic Resonance Imaging (MRI), Image Reconstruction, Accelerated Imaging, Constrained Reconstruction, Parallel Imaging, Inverse Problems, Numerical Optimization.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Parallel imaging as non-linear inverse problem

The signal acquired from a single voxel consists of the spin density ρ in the object's voxel volume and additional factors like relaxation, flow, diffusion and field inhomogeneities (*Bernstein et al.* 2004).

$$s_{\perp}(\vec{r},t) \propto \omega_0 \int_{\mathrm{FOV}} d^3r M_{\perp}(\vec{r}) B_{\perp}(\vec{r}) e^{-i(\omega_0-\Omega)t-\phi(\vec{r},t)} d\vec{r} \tag{1.1}$$

The field component $\vec{B}$ in equation 1.1 describes the spatial field distribution of the receiver coil.
The inducing component of the magnetic field originating from the voxel volume is called magnetization $\vec{M}$ and is responsible for the tissue depending image contrast.

$$\vec{M} = \vec{M}_0(1 - e^{-t/T_1})e^{-t/T_2^*} \tag{1.2}$$

$$\vec{M}_{\perp} = \vec{M}_0 \cdot e^{-t/T_2^*} \tag{1.3}$$

The signal change in the observed voxel depends on the tissue related T_1 time, which reduces the longitudinal component of the magnetization.
The effect of the tissue related T_2 decay is pronounced due to additional field influences. This results in the observed T_2^* time, which damps the transversal magnetization.
The magnetic field $\vec{M}_0$ in equation 1.2 represents the initial magnetization after

a 90° excitation pulse in the target volume, which is damped over time by the relaxation terms.
Because of the approximately 100-1000 times shorter T_2^* time the influence of the longitudinal magnetization can be neglected and the signal $s_\perp$ is mainly depending on the transversal component $\vec{M}_\perp$ (equation 1.3, *Haacke et al.* 1999).

$$\phi(\vec{r},t) = \phi_0(\vec{r}) - \gamma B(\vec{r})t = \phi_0(\vec{r}) - \theta_B(\vec{r},t) \tag{1.4}$$

The phase $\phi(\vec{r},t)$ of the signal oscillating with the Lamor-frequency ($\omega_L = \gamma B$), depends on the field angle θ_B relative to the initial angle ϕ_0 at $t = 0$, right after the excitation pulse.
It is therefore strong related to the configured MRI excitation coming from a RF (radio frequency) pulse or applied field gradient switches.
Additional factors like the larmor frequency ω_0 and distance-properites like d and r do also scale the signal amplitude.
The vector $\vec{r}$ is the spatial offset from the gradient isocenter, which defines the center of the FOV.

By using a quadrature amplitude demodulator (QAD) with frequency Ω the signal can be split into the corresponding transverse components, which equals the real and imaginary part of the acquired signal (equation 1.5).

$$s_\perp(t) = \mathrm{Re}(s_\perp(t)) + i \cdot \mathrm{Im}(s_\perp(t)) \tag{1.5}$$

Because the induced signal originating from the field $M_\perp$ is proportional to the spin density ρ and the coil sensitivities c_j equal the representative field $B_\perp(\vec{r})$ per receiver coil, equation 1.1 can be rewritten into equation 1.6,

$$s_j(k_x,k_y) = \int_{\mathrm{FOV}} \rho(x,y)c_j(x,y)e^{-i(k_x x + k_y y)}dxdy \tag{1.6}$$

where the kspace-trajectory vector $\vec{k}$ is split into the components k_x and k_y, which equal the corresponding phase shifts $\phi(x,t)$ and $\phi(y,G_y)$ that are applied

to encode the position of the acquired voxels in x and y direction (equation 1.7 and 1.8).
The gyromagnetic constant γ is specific for different kinds of nuclei. For example the hydrogen atom, which is mainly used in medical MRI, has a value of $2\pi \cdot 42.58 \frac{MHz}{T}$.

$$\phi(x,t) = k_x x = -\gamma x \int_0^t G_x(\tau) d\tau \tag{1.7}$$

$$\phi(y,G_y) = k_y y = -\gamma y \int_0^T G_y(\tau) d\tau \tag{1.8}$$

In parallel imaging the MRI signal is acquired from several coils surrounding the object with their corresponding coil sensitivity profiles.
To increase the speed of the acquisition of an entire image it is possible to acquire a reduced kspace and use the spacial coding of the coil sensitivities as additional information to prevent aliasing artifacts.
The spacial information can either be used to restore missing kspace lines prior to the Fourier transform (SMASH, GRAPPA), or in image space after the Fourier transform (SENSE, Pruessmann et al., 1999).
Because of the bi-linear structure it is not possible to single out the sensitivity profiles directly from the signal $s(t)$. One possibility is to perform a separate reference scan independent from the image acquisition.

1.1.1 IRGN background

The described IRGN method estimates the coil sensitivities c_1 from the acquired image with additional information about the field distribution in the image space.
To solve for the unknown object $\rho(\vec{r})$ function and the unknown sensitivity profiles $c_j(\vec{r})$ at once, a non-linear inversion technique like non-linear CG (*Hager et al.* 2006), BFGS (Broyden–Fletcher–Goldfarb–Shanno) (*BFGS algorithm* 1970) or a Gauss-Newton algorithm needs to be applied.

By using the iteratively regularized Gauss-Newton (IRGN) method it is possible to solve the MRI signal equation 1.9. It can be understood as a non-linear operator equation with an operator F, which maps the proton density and the coil sensitivity profiles to the measured data y. (Uecker, 2009)

$$y = F(x) \quad x = \begin{pmatrix} \rho \\ c_1 \\ \vdots \\ c_N \end{pmatrix} \tag{1.9}$$

The operator function $F(\rho, c) = P_k \mathcal{F} \rho c$, where P_k is the sampling pattern, which is one for the positions where the measured data exists and zero for non-sampled data positions.

In this work three different IRGN regularization strategies are described:

- Tikhonov (L2) regularization.
- Total variation (TV) regularization: Has positive effects for edge preservation and noise removal. But often leads to cartoon-like staircasing artifacts within inhomogeneous areas.(Knoll, 2011)
- Total generalized variation (TGV) regularization: Has the same advantages like TV in terms of edge preservation and noise removal and also suppresses staircasing artifacts.(Knoll, 2011)

To be able to solve the non-differentiable variational constraints (TV and TGV) for the IRGN algorithm a primal-dual extra gradient algorithm is used.

A drawback of the IRGN method is the increased computational burden, because it requires the calculation of a linear subproblem in several Gauss-Newton steps.

1.1.2 Gauss-Newton algorithm

The Gauss–Newton algorithm is a well known method to solve non-linear minimization problems.

The described method minimizes the squared L2-norm of the residuals, which can be interpreted as minimizing the squared euclidean length.

Starting with initial values x_0 the method updates the parameters x_n per iteration n by a calculated δx_n until a configured minimum residual norm $r_{norm}(x)$ is reached.

$$x_{n+1} = x_n + \delta x_n \tag{1.10}$$
$$r(x_n) = y - F(x_n) \tag{1.11}$$
$$r_{norm}(x_n) = \|y - F(x_n)\|_2^2 \tag{1.12}$$

Using the Taylor series expansion and stopping after the first derivative, equation 1.9 can be approximated into equation 1.13 which states the Gauss-Newton method to linearize the problem.

$$F(x) \approx F(x) + DF(x)\delta x \tag{1.13}$$
$$F(x_n + \delta x_n) \approx F(x_n) + DF(x_n)\delta x_n \tag{1.14}$$

$$F(x_{n+1}) = F(x_n) + DF(x_n)\delta x_n \tag{1.15}$$

$$\delta x_{n+1} = \arg\min_{\delta x_n} \|y - F(x_{n+1})\|_2^2 \tag{1.16}$$
$$\delta x_{n+1} = \arg\min_{\delta x_n} \|y - F(x_n) - DF(x_n)\delta x_n\|_2^2 \tag{1.17}$$

To solve the minimization problem 1.17 the derivative of the approximated residual with respect to the update gap needs to be calculated and set to zero.

$$\frac{\partial[y - F(x_n) - DF(x_n)\delta x_n]^2}{\partial[\delta x_n]} = 0 \tag{1.18}$$

$$2(y - F(x_n) - DF(x_n)\delta x_n)(-DF(x_n)^H) = 0 \tag{1.19}$$

$$y(-DF(x_n)^H) + F(x_n)DF_(x_n)^H + DF(x_n)\delta x_n)(DF(x_n)^H) \ = 0 \tag{1.20}$$

Equation 1.21 gives the Gauss-Newton formulation with lhs (left-hand-side) and rhs (right-hand-side) notation.

$$DF(x_n)^H DF(x_n)\delta x_n = DF(x_n)^H(y - F(x_n)) \tag{1.21}$$

Equation 1.22 gives the result for a Gauss-Newton update step.

$$\delta x_n = (DF(x_n)^H DF(x_n))^{-1} DF(x_n)^H \quad (y - F(x_n)) \tag{1.22}$$

1.1.3 Levenberg-Marquardt algorithm

The Levenberg-Marquardt algorithm extends the approximated Hessian $(DF(x)^H DF(x))$ by a positive definite regularization matrix (λI) to .

For large λ, this method is close to the steepest descent method 1.1.6, which is preferable at the beginning when the process is far off the solution.

For small λ, this method is close to the faster Gauss-Newton method 1.22, which is preferable, when the optimizer is close to the optimal point.

This way the algorithm becomes more robust, which means that in many cases it finds a solution even if the linearized equations are bad conditioned

(start very far off the final minimum) (*Levenberg–Marquardt algorithm* 2018).

$$\delta x_{n+1} = \arg\min_{\delta x_n} \|y - F(x_n) - DF(x_n)\delta x_n\|_2^2 + \lambda_n \|\delta x_n\|_2^2 \tag{1.23}$$

The update rule for δx stated in Equation 1.24 correspond to the squared minimization problem 1.23 with an additional Tikhonov L2-norm regularization therm (*Tikhonov regularization* 2018).

$$\delta x_n = (DF(x_n)^H DF(x_n) + \lambda_n I)^{-1} DF(x_n)^H \ (y - F(x_n)) \tag{1.24}$$

Another improvement is to apply the regularization not to the step δx, but to the result of the update with respect to the initial guess $x_n + \delta x - x_0$ (Uecker, 2009), which yields to the following minimization equation 1.25.

$$\delta x_{n+1} = \arg\min_{\delta x_n} \|y - F(x_n) - DF(x_n)\delta x\|_2^2 + \lambda \|x_n + \delta x_n - x_0\|_2^2 \tag{1.25}$$

Taking the derivative in respect to δx and setting the result equal to zero gives the result:

$$\frac{\partial[(y - F(x_n) - DF(x_n)\delta x)^2 + \lambda_n(x_n + \delta x - x_0)^2]}{\partial[\delta x]} = 0 \tag{1.26}$$

$$\begin{aligned} \frac{\partial[(y - F(x_n) - DF(x_n)\delta x)^2]}{\partial[\delta x]} &= 2(y - F(x_n) - DF(x_n)\delta x)DF(x_n)^H \\ \frac{\partial[\lambda_n(x_n + \delta x - x_0)^2]}{\partial[\delta x]} &= 2(x_n + \delta x - x_0)\lambda_n \end{aligned} \tag{1.27}$$

Applying the equations 1.27 to equation 1.26 gives the following result.

$$(y - F(x_n) - DF(x_n)\delta x)DF(x_n)^H \;+\; (x_n + \delta x - x_0)\lambda_n = 0 \tag{1.28}$$

$$yDF(x_n)^H - F(x_n)DF(x_n)^H - DF(x_n)\delta xDF(x_n) \; + \\ \lambda_n \delta x + \lambda_n(x_n - x_0) = 0 \tag{1.29}$$

$$DF(x_n)\delta xDF(x_n) + \lambda_n \delta x = \\ yDF(x_n)^H - F(x_n)DF(x_n)^H + \lambda_n(x_n - x_0) \tag{1.30}$$

The optimal solution of the IRGN algorithm for the update rule δx is stated in equation 1.31.

$$\delta x = (DF(x_n)^H DF(x_n) + \lambda_n I)^{-1} DF(x_n)^H (y - F(x_n)) + \lambda_n(x_n - x_0) \tag{1.31}$$

1.1.4 Determination of coil sensitivities

In general the coil sensitivities are rather smooth compared to the acquired object which does contain edges.
This a priori knowledge is used to achieve the desired regularization of the coil profiles.
The implemented algorithm is using a kspace weighting matrix to penalty height frequencies, which prefers coil intensity profiles (Uecker, 2009).

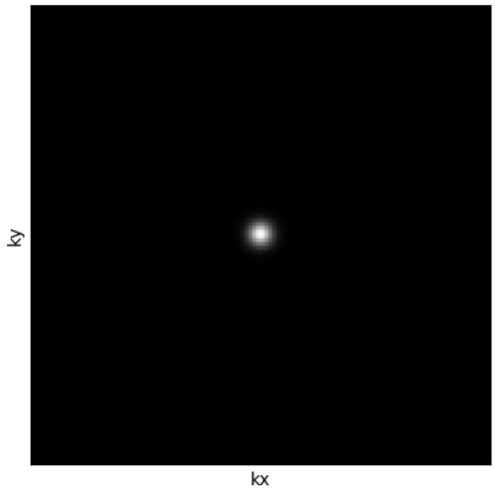


Figure 1.1: kspace intensity image of the applied weighting matrix W_{kspace}

Equation 1.32 and 1.33 show the applied operators for W and W^H.

$$W(\xi) = \mathcal{F}^{-1}(W_{kspace} \cdot \xi) \tag{1.32}$$
$$W^H(\xi) = W_{kspace} \cdot \mathcal{F}(\xi) \tag{1.33}$$

1.1.5 Differential of the function F(x)

For the calculation of an update step, the differential of the function $F(x)$ (equation 1.9) and its adjoint is needed.
Equation 1.34 and 1.35 shows $DF(x)$, which is the derivative of the operator function $F(x)$ and its adjoint $DF^H(x)$ (Uecker, 2009).

$$DF(x)\begin{pmatrix} \delta\rho \\ \delta c_1 \\ \vdots \\ \delta c_N \end{pmatrix} = \begin{pmatrix} P_k\mathcal{F}(\rho \cdot W(\delta c_1) + \delta\rho \cdot W(c_1)) \\ \vdots \\ P_k\mathcal{F}(\rho \cdot W(\delta c_N) + \delta\rho \cdot W(c_N)) \end{pmatrix} \tag{1.34}$$

$$DF^H(x)\begin{pmatrix}\delta y_1\\ \vdots \\ \delta y_N\end{pmatrix} = \begin{pmatrix}\sum_{i=1}^{N} W(c)_i^* \cdot \mathcal{F}^{-1}(P_k \delta y_i)\\ W^H(\rho^* \cdot \mathcal{F}^{-1}(P_k \delta y_1))\\ \vdots \\ W^H(\rho^* \cdot \mathcal{F}^{-1}(P_k \delta y_N))\end{pmatrix} \tag{1.35}$$

Where δy_i equals the difference between input rawdata y_i and the updated rawdata based on the current status of weighted coil sensitivities and image ρ (equation 1.36).

$$\delta y_i = \mathcal{F}(\rho \cdot W(c_i)) - y_i \tag{1.36}$$

The complex conjugated proton density and weighted coil sensitivities are written as ρ^* and c^*.

1.1.6 Steepest Descent algorithm

The steepest descent algorithm finds the local minimum of a function by taking steps proportional to the negative gradient direction of the function.
Therefore the algorithm is also known as gradient descent.

For large damping values λ the Levenberg-Marquardt formulation 1.24 equals the steepest descent update.

$$\delta x_n = (\lambda_n I)^{-1} DF(x_n)^H \quad (y - F(x_n)) \tag{1.37}$$

1.1.7 IRGN with L2 regularization

In chapter 1.1.2 the result of the minimization algorithm includes the Tikhonov L2-norm regularization.
The solution of the IRGN minimization algorithm is shown in figure 1.2.

Equation 1.25 can also be rewritten in the following form 1.39 with α and β being the regularization parameter for the coil sensitivities c and the spin-density ρ respectively.
$\mathcal{F}(x_n) - y$ equals the residuum δy_i (equation 1.36) per Gauss Newton step.

$$\delta x_{n+1} = \arg\min_{\delta x_n} \mathcal{J}(\delta\rho, \delta c) \tag{1.38}$$

$$\mathcal{J}(\delta\rho, \delta c) = \frac{1}{2}\left\|DF(x_n)\delta x + F(x_n) - y\right\|_2^2 + \frac{\alpha}{2}\left\|W(c_n + \delta c_n)\right\|_2^2 + \frac{\beta}{2}\left\|\rho_n + \delta\rho_n\right\|_2^2 \tag{1.39}$$

The result for the derivative with respect to ρ and c can be derived by applying equation 1.34 and 1.35 on 1.27.
The results are shown in equation 1.40 and 1.41.

$$\frac{\partial\mathcal{J}(\rho, c)(\delta\rho, \delta c)}{\partial\rho} = \sum_{i=1}^{N}[W(c)_i^* \cdot \mathcal{F}^{-1}(\mathcal{F}(\rho \cdot W(\delta c_i) + W(c_i) \cdot \delta\rho) + \delta y_i)] + \beta(\rho + \delta\rho - \rho^0) \tag{1.40}$$

$$\frac{\partial\mathcal{J}(\rho, c)(\delta\rho, \delta c)}{\partial c} = W^H[\rho^* \cdot \mathcal{F}^{-1}(\mathcal{F}(\rho \cdot W(\delta c_i) + W(c_i) \cdot \delta\rho) + \delta y_i] + \alpha(c + \delta c_i) \tag{1.41}$$

```
1  function IRGN(y, α, β, τ)
2  ρ ← 1
3  c, ρ_0 ← 0
4
5  repeat
6      δρ, δc ⇐ Solve(ρ, c, α, β, τ)
7      ρ = ρ + δρ
8      c = c + δc
9      α = max(α_min, α ∗ α_q)
10     β = max(β_min, β ∗ β_q)
11     tvits = min(tv_max, tvits ∗ 2)
12 until maxit
13 return ρ, c
```

Figure 1.2: IRGN algorithm with solve call *L2Solve*, *TVSolve* or *TGVSolve*

```
function L2Solve(ρ, c, α, β, τ)
δρ, δc, δ̂ρ, δ̂c ← 0

repeat
    η_ρ = ∂_ρ J(ρ, c)(δ̂ρ, δ̂c)
    η_c = ∂_c J(ρ, c)(δ̂ρ, δ̂c)
    δρ_old = δρ
    δc_old = δc
    δρ = δρ − τ ∗ η_ρ
    δc = δc − τ ∗ η_c
    δ̂ρ = 2 ∗ δρ − δρ_old
    δ̂c = 2 ∗ δc − δc_old
until tvits
return δρ, δc
```

Figure 1.3: Solution of the IRGN-L2 sub-problem

Figure 1.3 shows the calculation of the update rules $\delta\rho$ and δc for every Gauss Newton iteration of the IRGN algorithm shown in figure 1.2.

The primal descent step applies the τ scaled, negative gradient onto $\delta\rho$ and δc respectively. The calculation for the step length τ is shown in section 1.1.9. Also an extra gradient term $2 * x - x_{old}$ is used in every iteration.

1.1.8 TV and TGV regularization

The quadratic nature of the L2 regularization penalizes large values much stronger than small values originating from noise (Knoll, Kristian Bredies, et al., 2011).
To overcome this problem other regularization techniques like total variation (TV) or total generalized (TGV) can be used.
The algorithms for the TV and TGV subproblems are shown in figure 1.4 and 1.5.

$$TV(u) = \beta \int_{\Omega} \|\nabla u\|_1 \, dx \tag{1.42}$$

The regularization functional for the total variation is shown in equation 1.42. It changes the algorithm to a saddle point problem with the following form (1.43).

$$\min_{\delta\rho,\delta c}\max_{p_\beta} \widehat{\mathcal{J}}(\delta\rho,\delta c) + \langle \nabla_{xy}(\rho+\delta\rho), \widehat{p_\beta}\rangle - \mathbb{1}_{\|\cdot\|_\infty \le \alpha,\beta}(\widehat{p_\beta}) \tag{1.43}$$

$$\widehat{\mathcal{J}}(\delta\rho,\delta c) = \frac{1}{2}\left\|DF(x_n)\delta x + F(x_n) - y\right\|_2^2 + \frac{\alpha}{2}\left\|W(c_n+\delta c_n)\right\|_2^2 \tag{1.44}$$

The TV functional only applies changes to the minimization problem with respect to ρ. Therefore the derivative with respect to c has the same result like in the L2 regularization case.
The result for the derivative with respect to ρ is shown in equation 1.45.

$$\frac{\partial \mathcal{J}(\rho,c)(\delta\rho,\delta c)}{\partial\rho} = \sum_{i=1}^{N}[W(c)_i^* \cdot \mathcal{F}^{-1}(\mathcal{F}(\rho\cdot W(\delta c_i) + W(c_i)\cdot\delta\rho) + \delta y_i)] + \nabla_{xy}^T(\widehat{p_\beta}) \tag{1.45}$$

$$\widehat{p_\beta} = \frac{p_\beta + \sigma\nabla_{xy}(\rho + \widehat{\delta\rho}))}{max(1, \beta^{-1}\left\|p_\beta + \sigma\nabla_{xy}(\rho+\widehat{\delta\rho})\right\|_2)} \tag{1.46}$$

The nabla operator ∇_{xy} calculates point-wise the discrete differences.
It's adjoint operator ∇_{xy}^T equals the negative derivative $-div$ and calculates the point-wise transposed descrete finite differences (*Finite difference* 2018).

The term $p_\beta + \sigma\nabla_{xy}(\rho+\widehat{\delta\rho}))$ is called the dual ascent step.
It applies a positive gradient with step size σ on the dual variable p_β.
The parameter β in equation 1.46 controls how strong the dual update affects the primal gradient calculation.

```
function TVSolve(ρ, c, α, β, τ)
δρ, δc, δ̂ρ, δ̂c, p_β p̂_β ← 0

repeat
    η_ρ = ∂_ρ J(ρ, c)(δ̂ρ, δ̂c)
    η_c = ∂_c J(ρ, c)(δ̂ρ, δ̂c)
    δρ_old = δρ
    δc_old = δc
    δρ = δρ − τ ∗ η_ρ
    δc = δc − τ ∗ η_c
    δ̂ρ = 2 ∗ δρ − δρ_old
    δ̂c = 2 ∗ δc − δc_old
until tvits
return δρ, δc
```

Figure 1.4: Solution of the IRGN-TV sub-problem

A downside of the TV regularization is the assumption of piece-wise constant regions, which leads to a visual stair casing artifact (Knoll, 2011).
This problem can be solved by using the total generalized variation.
The TGV-functional changes the algorithm to a saddle point problem with the following form (equation 1.47).

$$\min_{\delta\rho,\delta c,v} \max_{\widehat{p_\beta},\widehat{p_{2\beta}}} \widehat{\mathcal{J}}(\delta\rho,\delta c) + \langle \nabla_{xy}(\rho+\delta\rho) - v, \widehat{p_\beta}\rangle + \langle \mathcal{E}v, \widehat{p_{2\beta}}\rangle - \mathbb{1}_{\|\cdot\|_\infty \le \alpha,\beta,2\beta}(\widehat{p_\beta}, \widehat{p_{2\beta}}) \quad (1.47)$$

$$\widehat{\mathcal{J}}(\delta\rho,\delta c) = \frac{1}{2}\|DF(x_n)\delta x + F(x_n) - y\|_2^2 + \frac{\alpha}{2}\|W(c_n+\delta c_n)\|_2^2 \quad (1.48)$$

The primal descent variable v balances the first and second derivative (see equation 1.49) in the TGV functional 1.47.
To calculate v a separate primal descent step per iteration of the sub-problem is included in the algorithm 1.5.

$$\mathcal{E}v = \frac{1}{2}(\nabla_{xy}v + \nabla_{xy}^{T}v) = (-div^2)\cdot v \tag{1.49}$$

Similar to the derivation of the minimization problem with TV regularization, also the TGV functional only has components depending on ρ.
The result of the derivative with respect to ρ and c is shown in equation 1.50 and 1.41.

$$\frac{\partial\mathcal{J}(\rho,c)(\delta\rho,\delta c)}{\partial\rho} = \sum_{i=1}^{N}[W(c)_i^* \cdot \mathcal{F}^{-1}(\mathcal{F}(\rho\cdot W(\delta c_i) + W(c_i)\cdot\delta\rho) + \delta y_i)] + \nabla_{xy}^{T}(\widehat{p_\beta}) \tag{1.50}$$

Due to the dependency of the dual ascent step on the primal descent variable v, equation 1.46 changes to 1.51.
Equation 1.52 calculates the dual ascent step depending on the regularization parameter β and the step length σ.

$$\widehat{p_\beta} = \frac{p_\beta + \sigma\nabla_{xy}(\rho + \widehat{\delta\rho}) - v)}{max(1, \beta^{-1}\left\|p_\beta + \sigma\nabla_{xy}(\rho + \widehat{\delta\rho}) - v\right\|_2)} \tag{1.51}$$

$$\widehat{p_{2\beta}} = \frac{p_{2\beta} + \sigma(\mathcal{E}v)}{max(1, 2\beta^{-1}\left\|p_{2\beta} + \sigma(\mathcal{E}v)\right\|_2)} \tag{1.52}$$

To calculate the primal dual variable v for the next iteration, an extra gradient term $2 * v - v_{old}$ is used.

```
function TGVSolve(ρ, c, α, β, τ)
δρ, δc, δ̂ρ, δ̂c, p_β, p̂_β, v, v̂ ← 0

repeat
    η_ρ = ∂_ρ J(ρ, c)(δ̂ρ, δ̂c)
    η_c = ∂_c J(ρ, c)(δ̂ρ, δ̂c)
    η_v = −div_xy p̂_2β − p̂_β
    δρ_old = δρ
    δc_old = δc
    v_old = v
    δρ = δρ − τ ∗ η_ρ
    δc = δc − τ ∗ η_c
    v = v − τ ∗ η_v
    δ̂ρ = 2 ∗ δρ − δρ_old
    δ̂c = 2 ∗ δc − δc_old
    v̂ = 2 ∗ v − δv_old
until tvits
return δρ, δc
```

Figure 1.5: Solution of the IRGN-TGV sub-problem

1.1.9 Lipschitz step-size

To calculate the Lipschitz constant L the implementation 1.6 uses some iterations of the power iteration method (*Power iteration* 2018) to approximately compute the norm of the partial Fréchet derivatives of the operator function F(x). Equation 1.53 shows an calculation step of the norm of DF(x), which is used to estimate the Lipschitz constant 1.54.
The initial values for x_ρ and x_c are random distributed.

$$\delta x_{i+1} = \frac{DF(x)\delta x_i}{\|DF(x)\delta x_i\|} \tag{1.53}$$

$$L = (DF(x)\delta x_{i+1})^T \cdot \delta x_{i+1} \tag{1.54}$$

Figure 1.6 shows an implementation of the algorithm used to calculate the step-size τ for the primal update term.
σ is the step-size for the dual update used for TV and TGV regularization.
The calculated step lengths σ and τ are in a range where the assumption $\sigma\tau L^2 < 1$ holds (Knoll, 2011).

```
% estimate operator norm using power iteration
x1  = rand(n,m); x2 = rand(n,m,nc);
[y1,y2] = M(x1,x2);
for i=1:10
    if norm(y1(:))~=0
        x1 = y1./norm(y1(:));
    else
        x1 = y1;
    end
    x2 = y2./norm(y2(:));
    [y1,y2] = M(x1,x2);
    l1 = y1(:)'*x1(:);
    l2 = y2(:)'*x2(:);
end
L = 2*max(abs(l1),abs(l2)); % Lipschitz constant estimate,
tau   = 1/sqrt(8+(L));      % primal step size
sigma   = 1/sqrt(8+(L));    % dual step size
```

Figure 1.6: Estimate operator norm using power iteration for estimating the Lipschitz constant and defining the primal and dual step-size (Lipschitz constant for gradient operator equals to 8)

1.1.10 Data initialization

As input data the algorithm receives the coil sensitivity data and regularization factors needed for the calculation.

The rawdata is normalized in the way, that the initial norm of the residuum (equation 1.12) has a value of 100.
This can be achieved with equation 1.55.

$$y = \frac{100}{\|data\|} \cdot data \tag{1.55}$$

1.1.11 Postprocessing

The applied postprocessing, which is shown in equation 1.56 scales the derived proton-density values with the RSS (root-sum-squared) of the calculated coil sensitivities.

To be able to compare different regularization techniques (Knoll, 2011) the result is also scaled by the normalization factor (see equation 1.55).

$$\rho_{irgn} = \rho \cdot \sqrt{\sum_{i=1}^{N} |W(c_i)|^2} \cdot \frac{100}{\|data\|} \tag{1.56}$$

1.2 CUDA

Because of the nature of a graphics processing unit (GPU) to handle and calculate big datasets concurrently, it is also used in computational medicine.

The compute unified device architecture (CUDA) by Nvidia offers an application programming interface (API) which allows software developers the use of Nvidia-GPUs as general purpose processing unit.
This approach is also known as GPGPU.

Modern computer architectures are classified by the Flynn‘s Taxonomy.
The definitions (1.1) are based upon the possible number of concurrent instructions and data streams in a processing unit (*Flynn's taxonomy* 2018).

SISD	single-instruction, single-data	e.g. single core CPU
MIMD	multiple-instruction, multiple-data	e.g. multi core CPU
SIMD	single-instruction, multiple-data	e.g. data-based parallelism
MISD	multiple-instruction, single-data	e.g. fault-tolerant computers
SIMT	single-instruction, multiple-threads	Is a combination of SIMD with multi threading, which is used in modern parallel computing (CUDA).

Table 1.1: Extended Flynn‘s Taxonomy

Since the goal for a CPU is to get the best performance for a single heavy weight thread, it's latency oriented design concentrates following factors:

- Big data caches
- Low latency arithmetic units
- Complex Control Logic: Branch prediction, Out-of-order-execution

The purpose of a GPGPU is to get the best performance for a lot of simple threads, which leads to an throughput oriented design:

- Small caches
- Hide latency with computation
- In-order execution without branch prediction

- Issue the same command to multiple cores

Figure 1.7 shows the dependency of reachable speedup by parallelizing sequential instructions.
Amdahl's definition for the achievable speedup s is a function (1.57) depending on the number of processors p and the fraction of parallelize-able code N *Amdahl's law* 2018.

$$s = \frac{1}{(1-p) + \frac{p}{N}} \tag{1.57}$$

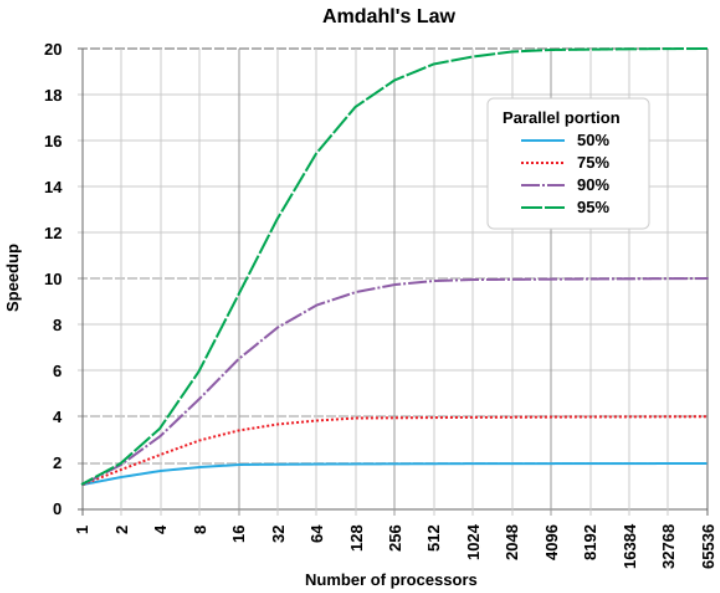


Figure 1.7: Amdahl's Law

1.2.1 CUDA environment

A CUDA program calls kernel functions, which execute a set of 32 parallel threads with the same instructions.

A unit of 32 threads is called a warp (NVidia, 2012).

Each thread is running on a scalar processor (*Scalar processor* 2017) with very fast memory access to its own registers and local memory areas (see figure 1.8).
A streaming multiprocessor (SM) consists of several scalar processors and a fast L1 cache with a shared memory space for inter-block communication.

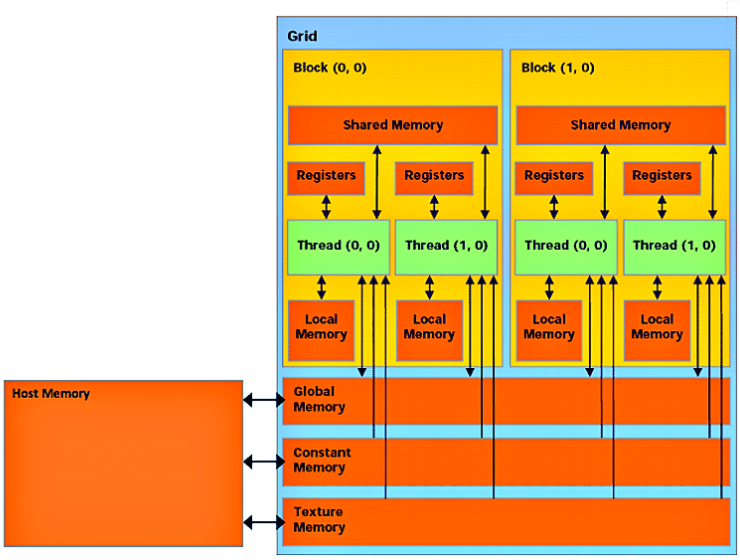


Figure 1.8: CUDA memory model

An example call of a kernel function with defined sizes for blocks per grid and threads per block is shown in figure 1.9.

```
1 functionName<<<blocksPerGrid, threadsPerBlock>>>(output, input);
```

Figure 1.9: Example for a CUDA device function call with defined block and grid size

Because the IRGN and Agile implementation focuses on fast matrix and vector calculations, it is crucial to use as many threads with fast memory access per calculation as possible.

A grid element can only share its results into the global memory space after a kernel-wide global synchronization, which makes the calculation slow and is therefore used for other application concepts.

To utilize as many threads with a fast memory communication pipeline as possible, the correct number of blocks per grid and number of threads per block needs to be calculated.
Equation 1.58 shows the calculation of the needed number of blocks $nbBlocksPerGrid$ for a maximal number of threads per block $nbThreadsPerBlock$ and given data elements N.

$$nbBlocksPerGrid = \frac{N + nbThreadsPerBlock - 1}{nbThreadsPerBlock} \tag{1.58}$$

Since the release of devices with compute compatibility 3.0 the maximal number of threads per block for dimension x is 1024 (*CUDA C Programming Guide* 2018).
Another limiting factor is the size of the shared memory in a streaming multiprocessor, which executes the instructions of a block (see figure 1.10).

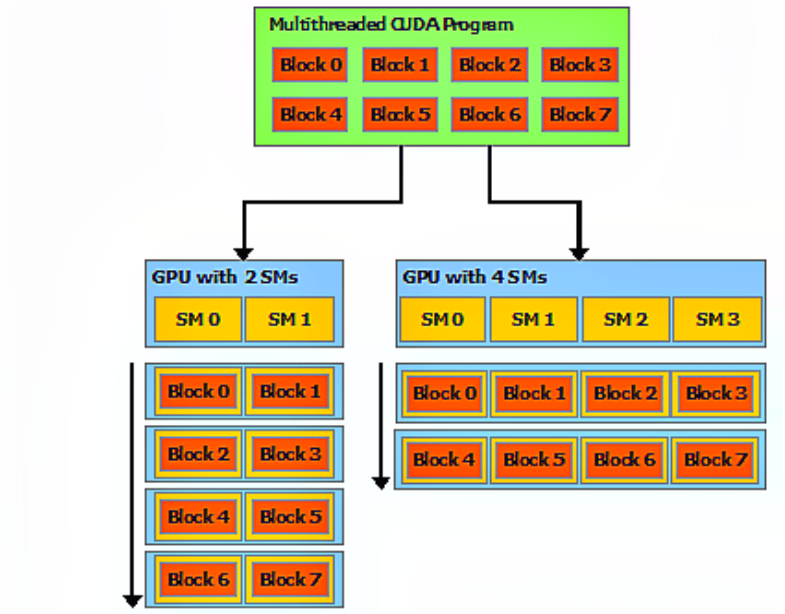


Figure 1.10: CUDA's automatic scalability for multi streaming multiprocessor architectures

```
template <typename floatType, typename complexType>
void pattern( floatType           * z,
              complexType   const * x,
              unsigned      const & size )
{
  unsigned nbBlocksPerGrid = (size + maxNumThreadsPerBlock - 1)
                           / maxNumThreadsPerBlock;
  unsigned nbThreadsPerBlock = maxNumThreadsPerBlock;

  patternGPU<floatType> <<<nbBlocksPerGrid, nbThreadsPerBlock>>>(
                                                          z,
                                                          x,
                                                          size );
}
```

Figure 1.11: CUDA device function *patternGPU* call with defined block and grid size

Figure 1.11 shows the call of the kernel function *patternGPU* with defined number of blocks and threads.

The purpose of the multiprocessor control unit is to schedule the warp executions, which strongly depends on their execution time.

To gain high performance the first step is to maximize the overall memory throughput for the application by minimizing data transfers with low bandwidth (*CUDA C Programming Guide* 2018).

Another important performance factor is to maximize the instruction throughput with following measurements:

- use of arithmetic instructions with high throughput: intrinsic functions, single-precision.
- minimize divergent warps caused by control flow instructions.
- optimize synchronization points (*__syncthreads()*).

To declare a host side function execution on the GPU, the keyword __global__ needs to be used.
Each thread that executes the kernel function is given an unique thread index number in a block, that is accessible through the built-in $threadIdx$ variable.
Also each block in a grid has an identification number, which is defined via the built-in variable $blockIdx$.
To access the currently running thread, it's global id $threadId$ can be calculated using the defined variables $threadIdx$, $blockIdx$ and the actual block and grid dimensions ($blockDim$, $gridDim$).
An example for the calculation of the thread ID in an one dimensional block can be seen in equation 1.59.

$$threadId = blockDim.x * blockIdx.x + threadIdx.x; \tag{1.59}$$

Figure 1.12 shows the source code of the kernel function *patternGPU*.
To avoid an out of range access of the given data-pointer, the upper boundary for the threadId must be set to the given data size N.

```
template <typename floatType, typename complexType>
__global__ void patternGPU(floatType          * z,
                           complexType  const * x,
                           unsigned     const & N)
{
  unsigned threadId = blockDim.x * blockIdx.x + threadIdx.x;

  if (thread_id < N)
  {
    floatType norm = agile::norm( x[threadId] );
    z[threadId] = (sqrt(norm) > 0) ? floatType(1) : floatType(0);
  }
}
```

Figure 1.12: CUDA global device function *patternGPU*

cuBLAS

The cuBLAS library is part of the AGILE environment and used for the IRGN calculation.
It is an addition to the AGILE low-level implementation of basic matrix and vector calculations.

Nvidia's cuBLAS library is a parallelized implementation of BLAS (Basic Linear Algebra Subprograms), which provides standard building blocks to perform basic vector and matrix operations (*BLAS (Basic Linear Algebra Subprograms)* 2018).

The functionality is divided into three levels, which correspond to the complexities of the algorithms.
Level 1 performs scalar, vector and vector-vector operations, Level 2 matrix-vector operations and Level 3 performs matrix-matrix operations.

The Agile classes for matrix (GPUMatrix 2.2.1) and vector (GPUVector 2.2.1) calculations define wrapper functions for the second version of cuBLAS (cuBLAS_v2).

cuFFT

The Nvidia's cuFFT implementation is based on the Cooley-Tukey and Bluestein algorithm for the calculation of the Fast Fourier Transform.

The implementation uses a divide-and-conquer algorithm with unrolled recursion, loops and conditionals *cuFFT :: CUDA Toolkit Documentation* 2018.

The combination of the floating-point power and parallelism of the GPU with this technique, optimizes the efficiency to calculate the discrete Fourier transform.

The cuFFT library supports the following features:

- Complex and real-valued 1D, 2D and 3D transforms.
- Highly optimization for input sizes of the powers of two.
- Floating point numbers up to double-precision (64-bit).
- Streamed and batched execution, enabling asynchronous computation and data movement.

1.3 Agile

The functionality for the CUDA powered implementation of the IRGN-T(G)V algorithm is based on the AGILE library (Freiberger et al., 2013).
AGILE offers an environment for Linear and non-linear image reconstruction using the GPU and is an open source library.
It is implemented with C++ and the project is managed with CMake.

2 Methods

The CUDA powered IRGN-T(G)V application got developed on a PC workstation equipped with a CUDA-powered GPU.
A 64bit linux operating system has been chosen as compile target.
The used PC or server operating system is Ubuntu linux version 16.04.5 LTS - cenial.
The linux operating system can be downloaded from the official ubuntu homepage (https://www.ubuntu.com).

2.1 Getting started

To be able to compile, link and execute the application, the Nvidia driver for the GPU need to be installed on the system.

2.1.1 Qt creator

The Qt Creator IDE can be downloaded and installed via the YaST package manager. The used version is 2.6.2.

2.1.2 CUDA

The CUDA Framework can be downloaded from the Nvidia homepage https://developer.nvidia.com/cuda-downloads.
The used toolkit version is v7.5.

According to the used CUDA version, also the used compiler needs to be updated according to the Nvidia reference.

2.1.3 Agile

The Agile library can be downloaded from a GIT repository which is located at https://github.com/IMTtugraz/AGILE.

2.2 Agile additions

The agile library has been extended with low and high-level code for the calculation of the IRGN algorithm.

High-level extensions effect the implementation of the *GPUMatrix*, *GPUVector*, *GPUComplex* classes.
The Agile low-level implementation of the cuBLAS_v2 functionality is located in the file *gpu_matrix.ipp*.
GPU kernel implementations are located in the file *gpu_vector.ipp*.

2.2.1 Matrix functions

The following GPUMatrix methods wrap functions of the second cuBLAS version (cuBLAS_v2) and the agile low-level implementation:

- Scale a matrix A with a scalar $alpha$ and copy the result to B.

```
template <typename TType>
void scale( const typename to_real_type<TType>::type& alpha,
            const GPUMatrix<TType>& A,
            GPUMatrix<TType>& B );
```

- This method calculates the phase of complex matrix elements by using the low-level *phaseVector* method located in *gpu_vector.ipp*.

```
template <typename TType1, typename TType2>
inline void phase( const GPUMatrix<TType1>& X,
                   GPUMatrix<TType2>& Y );
```

- *fftshift* and *ifftshift* perform a shift of the zero-frequency component to center the spectrum of a kspace data matrix.
  Both are wrapper functions for the low-level implementation located in the *gpu_vector.ipp* file.

```
template <typename TType1>
void fftshift(TType1* x, unsigned rows, unsigned cols);

template <typename TType1>
void ifftshift(TType1* x, unsigned rows, unsigned cols);
```

- Complex or real square-root low-level calculation

```
template <typename TType>
__global__ void sqrt_GPU( const TType* x, TType* y, unsigned
    size )
{
  unsigned thread_id = blockDim.x * blockIdx.x + threadIdx.x;
  if (thread_id < size)          //while
  {
    y[thread_id] = agile::cusqrt(x[thread_id]);
    //thread_id += blockDim.x*gridDim.x;
  }
}
```

Square root of a complex number

The Agile library has been extended with the functionality to calculate the square-root of a complex number (Rabinowitz, n.d.).
The calculation is shown in equation 2.1.

$$\sqrt{x+iy} = \sqrt{\frac{\sqrt{x^2+y^2}+x}{2}} + sign(y) * i \sqrt{\frac{\sqrt{x^2+y^2}-x}{2}} \tag{2.1}$$

The compiler flag __inline__ is used to reduce instruction fetch stalls during a warp execution. It should only be used in conjunction with small functions.

The method *complexSqrt()* is compiled for the device (GPU) and the host (CPU) by using the flags *__device__* and *__host__*.

- Implementation of the complex square-root calculation.

```
// class-wide definition:
#define DEVICEHOST __inline__ __device__ __host__

DEVICEHOST GPUComplex complexSqrt() const
{
  TType vz=1;
  if(y<0)
    vz=-1;
  TType real = TType(sqrtf((sqrtf(x*x+y*y)+x)/2));
  TType imag = TType(vz*sqrtf((sqrtf(x*x+y*y)-x)/2));

  return GPUComplex( real, imag );
}
```

2.2.2 Vector functions

- Low-level phase calculation of the entries in the complex vector x.

```
void phaseVector( const TType1* x,
                  TType2* y,
                  unsigned size );
```

- Calculation of the pattern vector as described in 2.2.3.

```
void pattern( const TType* x,
              typename to_real_type<TType>::type* z,
              unsigned size );
```

- The *get_content* method performs a hard-copy in the global memory of the given data pointer x_{data} with defined sizes.

```
void get_content( const TType* x_data,
                  unsigned rows,
                  unsigned cols,
                  unsigned row_offset,
                  unsigned col_offset,
                  TType* z,
                  unsigned z_rows,
                  unsigned z_cols );
```

2.2.3 FFT class

The FFT class provides methods to transform a matrix from the kspace (Fourier domain) to the image space and vice versa.
The forward and inverse fast Fourier transform is computed by the NVIDIA® CUDA™ cuFFT library (see chapter 1.2.1).

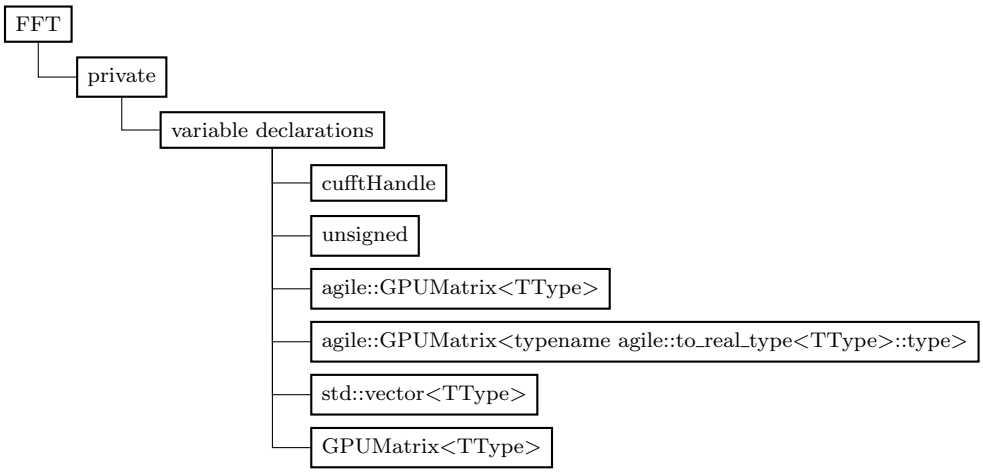


Figure 2.1: FFT private Class structure

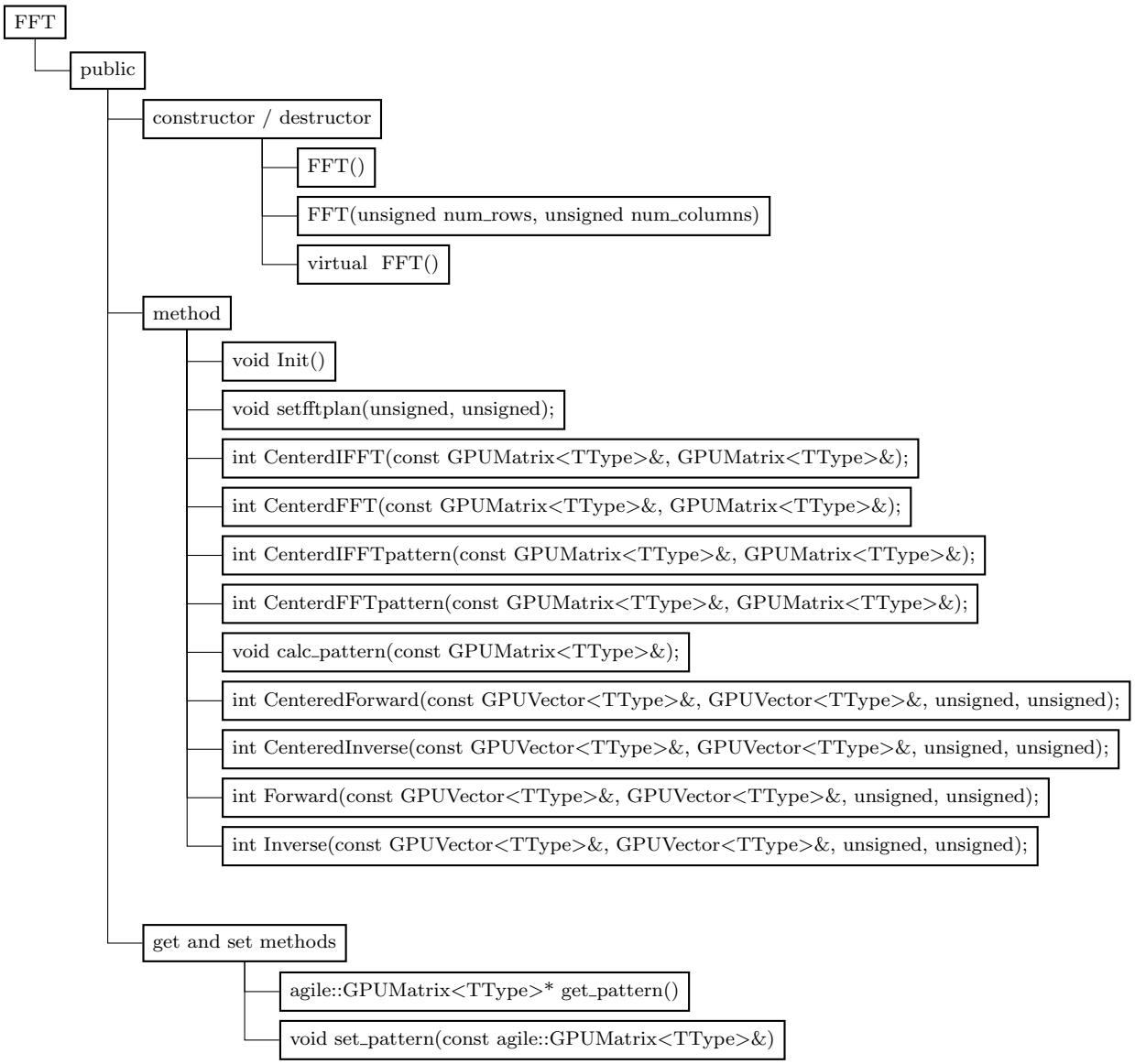


Figure 2.2: FFT public Class structure

Description of methods

The methods CenteredFFT and CenteredIFFT are used to shift between centered and not centered Fast Fourier Transform.
This considers the matrix element [0,0] to be the DC-part in the center of the kSpace.

CenterdFFTpattern and CenterdIFFTpattern apply a calculated pattern (calc_pattern) in the fourier-domain.

- Default constructor.

```
FFT()
```

- Constructor with fft-plan definition for given size.

```
FFT(unsigned num_rows,
    unsigned num_columns)
```

- Destructor destroys the predefined fft-plan.

```
virtual ~FFT()
```

- The init-method initializes the ones_complex_vec_cpu_nxns_ matrix with TType value 1, which is needed for pattern generation.

```
void Init()
```

- The cuFFT API provides a simple configuration mechanism, which is called *plan*, that uses internal building blocks to optimize the transformation for the given configuration.
  setfftplan creates a plan by wrapping the cufftPlan2d method.

```
void setfftplan(unsigned num_rows,
                unsigned num_columns);
```

- Calculates a pattern matrix from given in_mat and saves the result in the member variable pattern_complex with value type TType.
  The calculated pattern value is 1 (for complex value: 1+0i), if the absolute input value is greater than 0.

```
void calc_pattern(const GPUMatrix<TType>& in_mat);
```

- A pre-calculated pattern matrix can be set or get via the following public methods.

```
agile::GPUMatrix<TType>* get_pattern()
void set_pattern(const agile::GPUMatrix<TType>& pattern_mat);
```

- Calculates the centered fourier transform of in_mat.
  The result is given by reference in new out_mat matrix.

```
int CenterdIFFT(const GPUMatrix<TType>& in_mat,
                GPUMatrix<TType>& out_mat);}
int CenterdFFT(const GPUMatrix<TType>& in_mat,
                GPUMatrix<TType>& out_mat);
```

- Calculates the centered fourier transform of in_mat with applied pattern.
  The result is given by reference in new out_mat matrix.

```
int CenterdIFFTpattern(const GPUMatrix<TType>& in_mat,
                       GPUMatrix<TType>& out_mat);
int CenterdFFTpattern(const GPUMatrix<TType>& in_mat,
                      GPUMatrix<TType>& out_mat);
```

- Calculates the fourier transform of a given input vector in_vec with the predefined fft-plan.
  A start offset of the data-pointer can be set via in_offset and out_offset.

```
int Forward(const GPUVector<TType>& in_vec,
            GPUVector<TType>& out_vec,
            unsigned in_offset = 0,
            unsigned out_offset = -1);
int Inverse(const GPUVector<TType>& in_vec,
            GPUVector<TType>& out_vec,
            unsigned in_offset = 0,
            unsigned out_offset = -1);
```

- Calculates the centered fourier transform of a given input vector in_vec with the predefined fft-plan.
  A start offset of the data-pointer can be set via in_offset and out_offset.

```
int CenteredForward(const GPUVector<TType>& in_vec,
                    GPUVector<TType>& out_vec,
                    unsigned in_offset = 0,
                    unsigned out_offset = -1);
int CenteredInverse(const GPUVector<TType>& in_vec,
                    GPUVector<TType>& out_vec,
                    unsigned in_offset = 0,
                    unsigned out_offset = -1);
```

2.2.4 KSpaceFOV class

To prevent backfolding artifacts in the frequency encoded direction, the FOV (field of view) has to be larger than the imagined specimen.
Due to the fast sampling of the frequency direction it is common to configure a rectangular FOV.
With the help of the KSpaceFOV class it is possible to resize a kspace matrix by cropping the data matrix in the image domain.

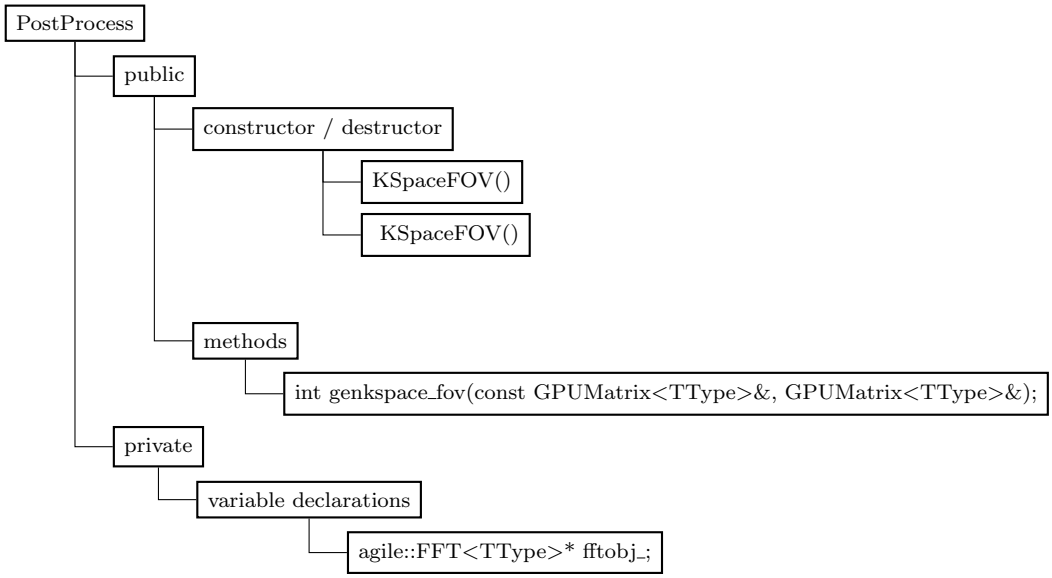


Figure 2.3: KSpaceFOV public Class structure

Description of methods

- The default constructor initializes a new FFT object.

```
KSpaceFOV()
```

- Destructor frees the FFT object and releases its resources.

```
virtual ~KSpaceFOV()
```

- The method genkspace_fov resizes the FOV for the input matrix in_mat and returns the result by reference in out_mat.
  The resulting matrix has a squared dimension with the size defined by the number of input rows (phase direction).

```
int genkspace_fov(const GPUMatrix<TType>& in_mat,
                  GPUMatrix<TType>& out_mat);
```

2.2.5 PostProcess class

The PostProcess Class consists of methods to get the real or imaginary part from a complex valued matrix.
Methods to calculate the absolute or phase value are also included.

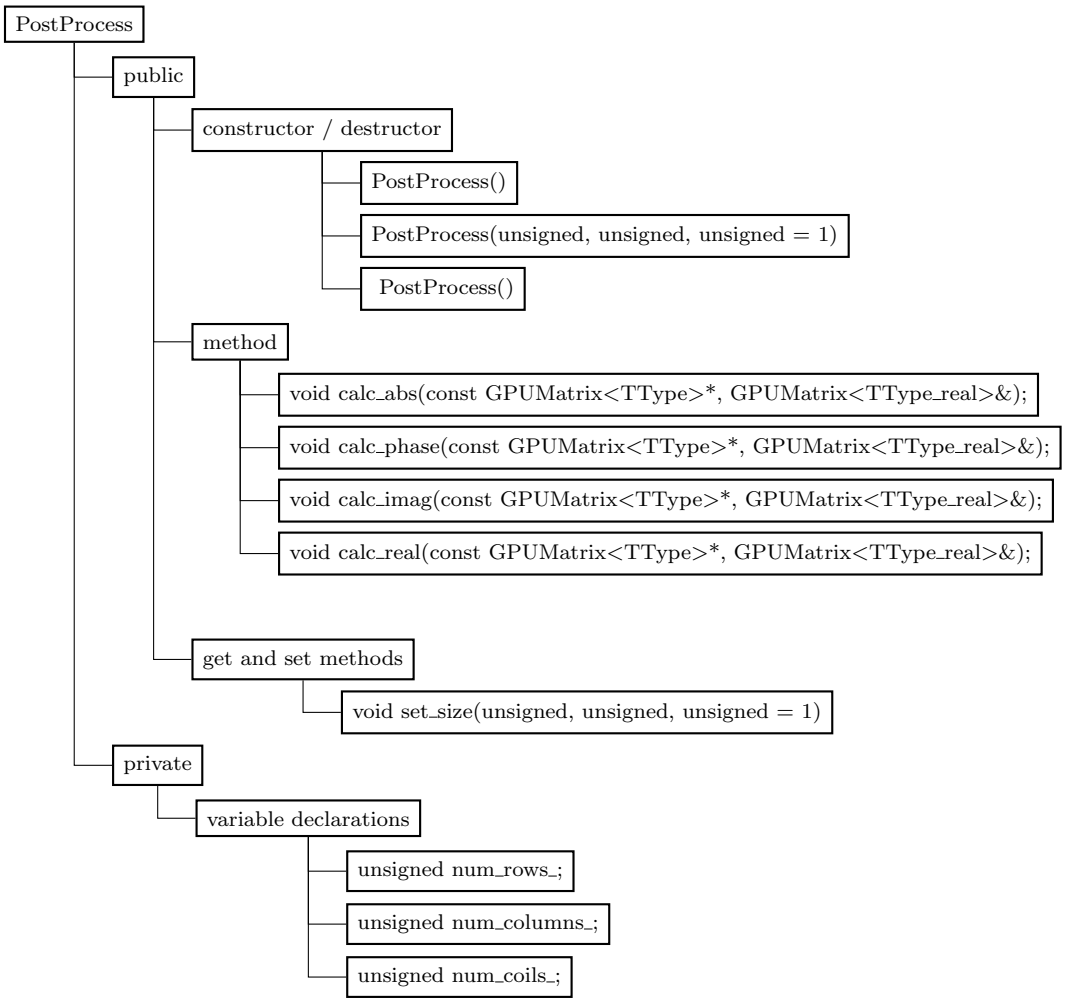


Figure 2.4: PostProcess public Class structure

Description of methods

- The default constructor initializes a PostProcess object.

```
PostProcess()
```

- Constructor to initialize a PostProcess object with given dimensions.

```
PostProcess(unsigned num_rows,
            unsigned num_columns,
            unsigned num_coils = 1)
```

- Destructor

```
virtual ~PostProcess()
```

- Calculates the absolute values of an input matrix and returns a reference to the resulting matrix.

```
void calc_abs(const GPUMatrix<TType>* in_mat,
              GPUMatrix<TType_real>\& out_mat);
```

- Calculates the phase values of an input matrix and returns a reference to the resulting matrix.

```
void calc_phase(const GPUMatrix<TType>* in_mat,
                GPUMatrix<TType_real>\& out_mat);
```

- Returns a reference to a matrix with the imaginary values of an input matrix.

```
void calc_imag(const GPUMatrix<TType>* in_mat,
               GPUMatrix<TType_real>\& out_mat);
```

- Returns a reference to a matrix with the real values of an input matrix.

```
void calc_real(const GPUMatrix<TType>* in_mat,
               GPUMatrix<TType_real>\& out_mat);
```

2.3 IRGN-L2/T(G)V implementation

The implementation of the IRGN algorithm in C++ is based on the Matlab code written by Clason and Bredies (*Clason, Bredies* 2011).
To simplify the comparison of the implementation between the Matlab and C++, the method (function) names are equal and start with a capital letter (in contrast to usual coding standards).

The IRGN algorithm (figure 1.2) is implemented in the *Iteration* method located in the IRGN class.
The update rules $\delta\rho$ and δc for every Gauss Newton iteration of the IRGN algorithm can be calculated by using one of the three implemented sub-problems.
The solver for the IRGN sub-problem with a L2 regularization is shown in figure 1.3.
The implemented counterpart in C++ is implemented in the L2Solve class.
Implementations for the IRGN sub-problems with TV (figure 1.4) and TGV (figure 1.5) regularization can be found in the classes TVSolve and TGVSolve respectively.

2.3.1 Structure

The abstract base class IRGN implements methods to initialize and compute the IRGN algorithm.
The ability to choose between different solvers with L2, TV or TGV regularizers is implemented via the object oriented polymorphism mechanism.
The classes L2Solve, TVSolve and TGVSolve inherit from the abstract base class IRGN and override the pure virtual method Solve(...).

Figure 2.5 shows the inheritance structure.

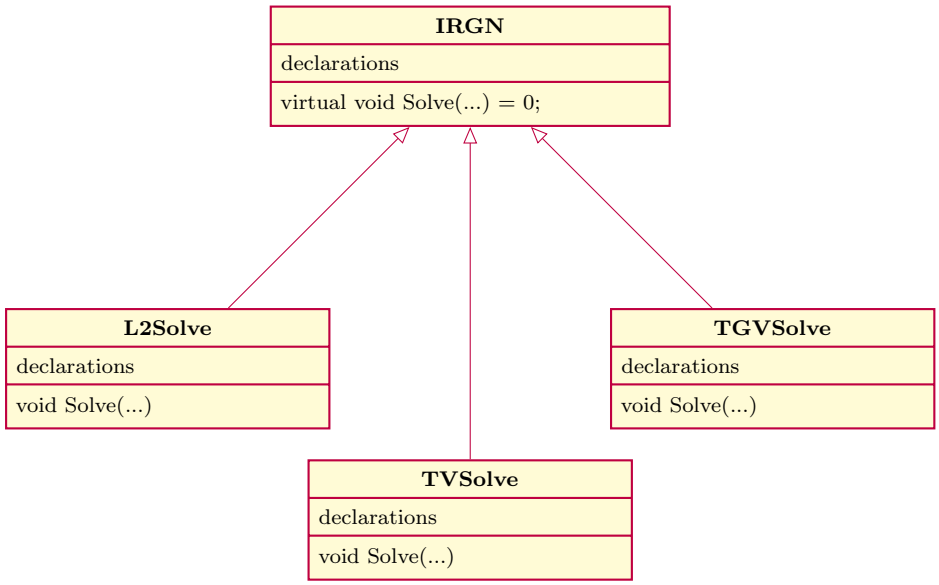


Figure 2.5: IRGN Structure

2.3.2 IRGN parameter configuration

Figure 2.6 shows the IRGN_params structure, which declares the IRGN algorithm configuration.

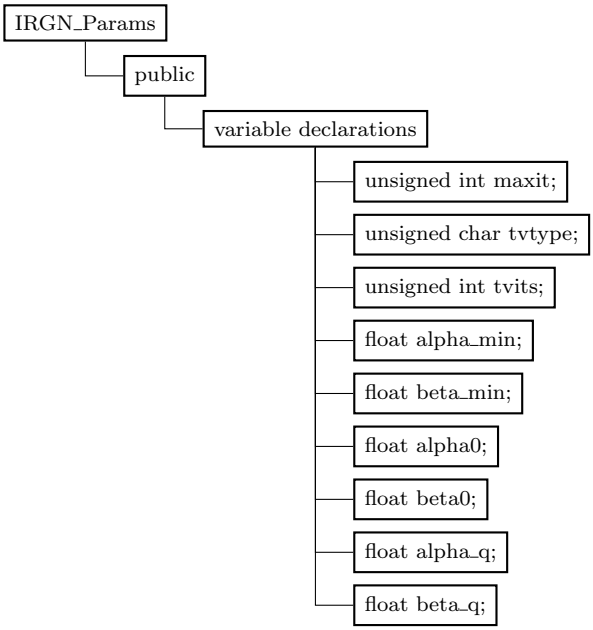


Figure 2.6: IRGN_Params structure

2.3.3 IRGN class

Figure 2.7, 2.8 and 2.9 show the declaration structure of the IRGN class.

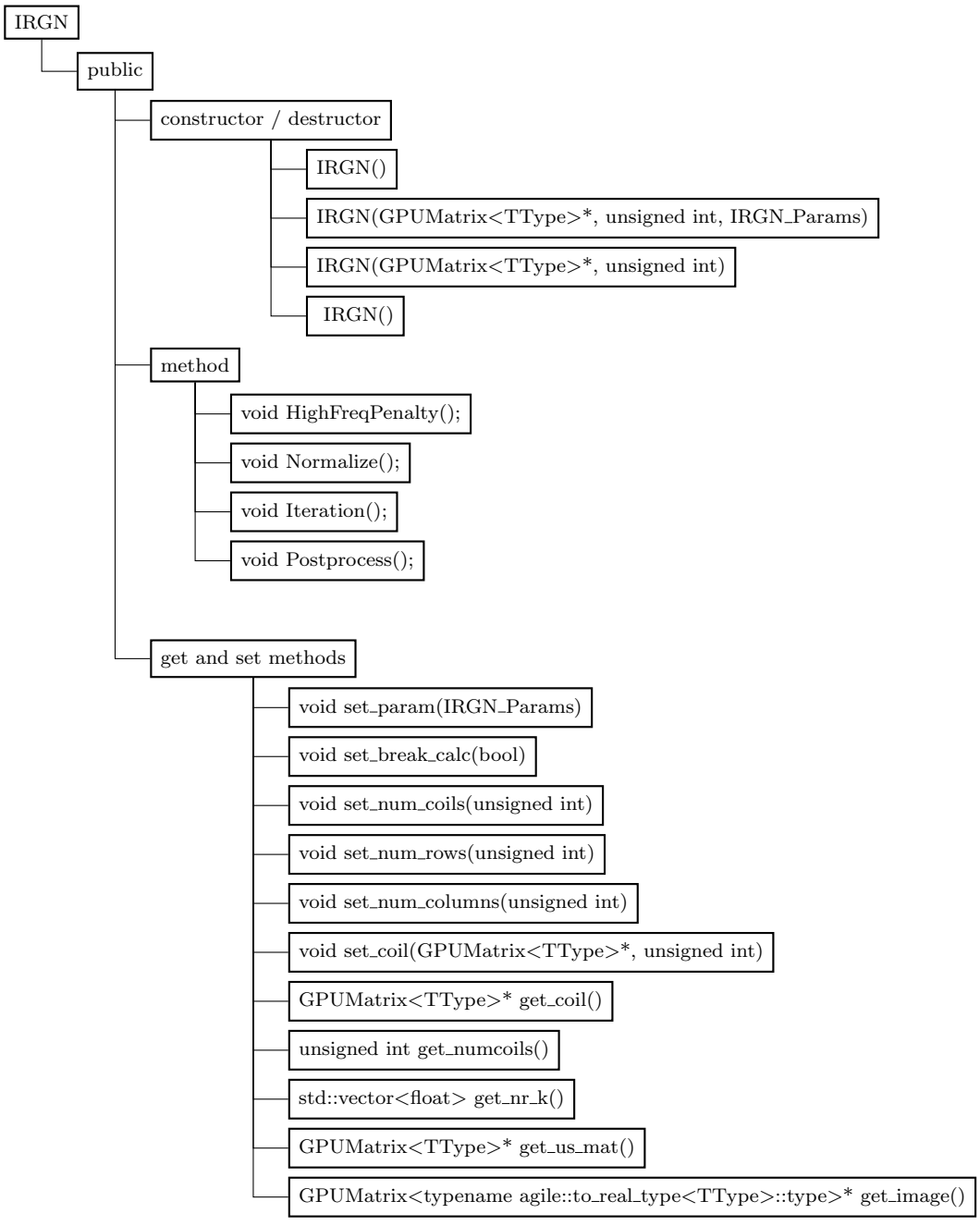


Figure 2.7: IRGN Class Structure Public

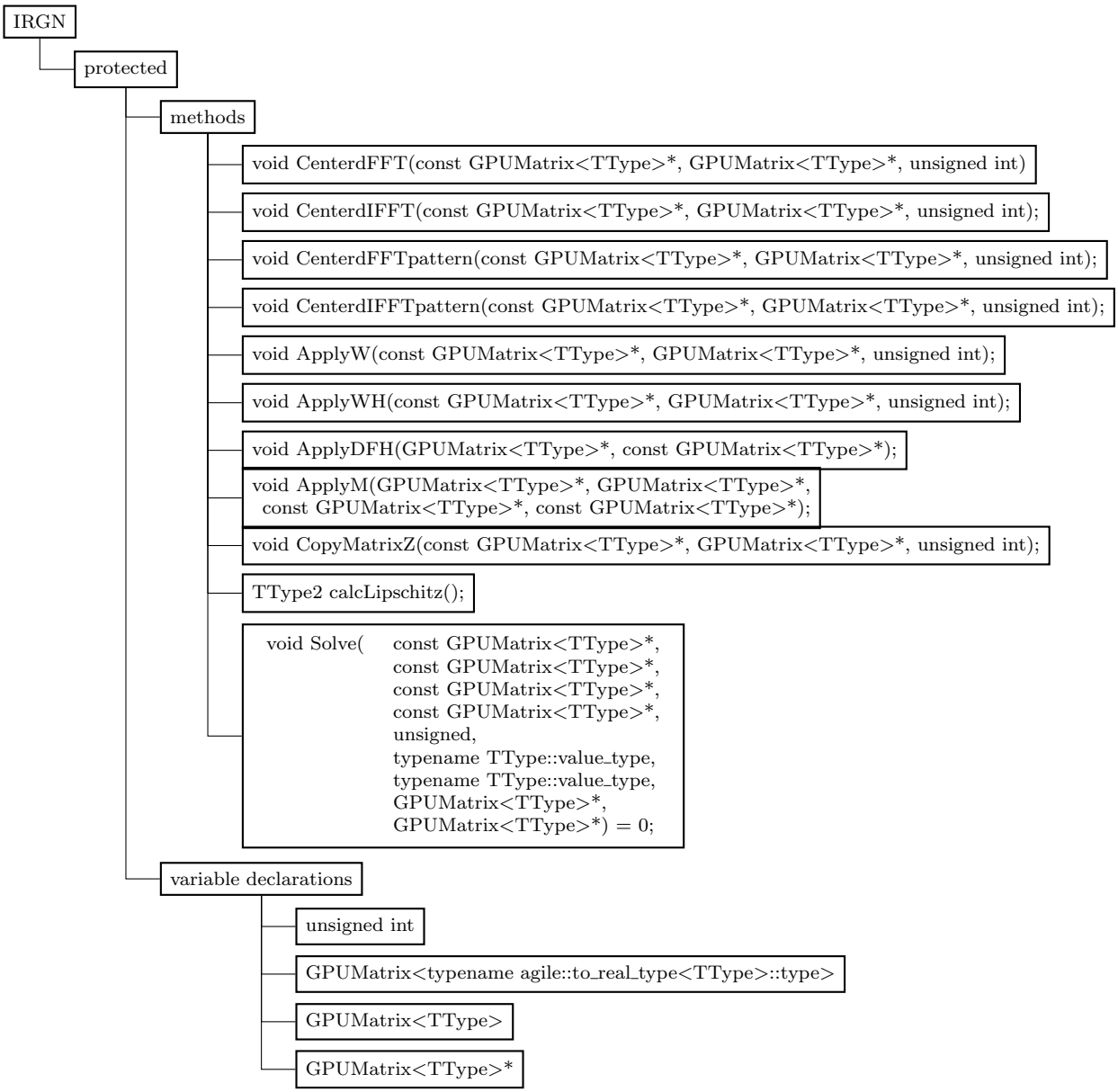


Figure 2.8: IRGN Class Structure Protected

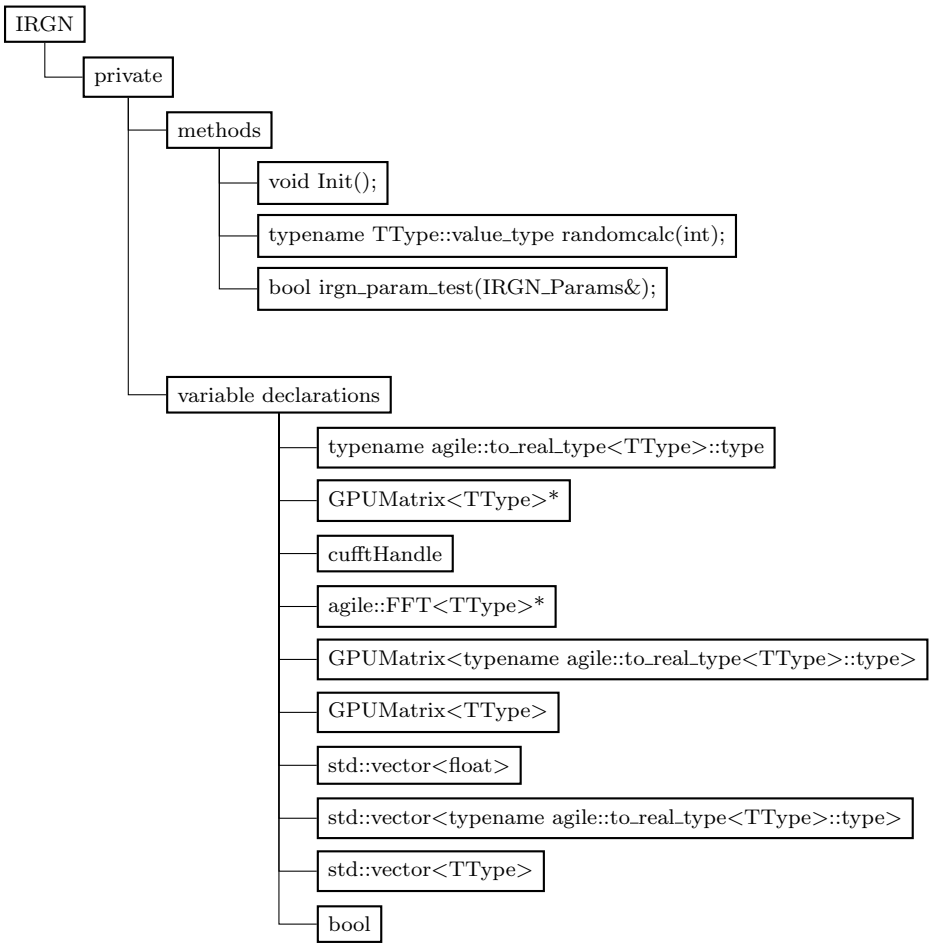


Figure 2.9: IRGN Class Structure Private

Description of methods

Public methods

- The default constructor creates an empty IRGN object.

```
1 IRGN()
```

- This constructor initializes an IRGN object with defined data and configuration parameters.

```
IRGN(GPUMatrix<TType>* coil,
     unsigned int num_coils,
     IRGN_Params param)
```

- Constructor to initialize an IRGN object with defined data.

```
IRGN(GPUMatrix<TType>* coil,
     unsigned int num_coils)
```

- Destructor frees used resources.

```
~IRGN()
```

- Calculates the high frequency penalty 'weight' matrix.

```
void HighFreqPenalty()
```

- Normalizes the data matrices of the provided coil data.

```
void Normalize()
```

- Starts the calculation of the Gauss-Newton algorithm.

```
void Iteration()
```

- Post-processes the calculated data.

```
void Postprocess()
```

- Sets the IRGN parameters for the declared IRGN_Params structure.

```
void set_param(IRGN_Params param)
```

- Force the iteration of a Gauss-Newton step to break.

```
set_break_calc(bool break_calc)
```

- Sets the number of coils.

```
void set_num_coils(unsigned int num_coils)
```

- Sets the number of rows.

```
void set_num_rows(unsigned int num_rows)
```

- Sets the number of columns.

```
void set_num_columns(unsigned int num_columns)
```

- Sets a GPUMatrix<TType> pointer to the coil data with given number of columns.

```
void set_coil(GPUMatrix<TType>* coil,
              unsigned int num_coils)
```

- Returns a GPUMatrix<TType> pointer to the coil data.

```
GPUMatrix<TType>* get_coil()
```

- Returns the number of coils.

```
unsigned int get_numcoils()
```

- Returns the actual number of iterations.

```
std::vector<float> get_nr_k()
```

- Returns a GPUMatrix<TType> pointer to the calculated result.

```
GPUMatrix<TType>* get_us_mat()
```

- Returns a reference to the image matrix with the stored absolute values.

```
GPUMatrix<typename agile::to_real_type<TType>::type>*
    get_image()
```

Protected methods

- Calculates the centered fast Fourier transform of the input matrix with defined size.

```
void CenterdFFT(const GPUMatrix<TType>* in_mat,
                GPUMatrix<TType>* out_mat,
                unsigned int num_z)
```

- Calculates the centered inverse fast Fourier transform of the input matrix with defined size.

```
void CenterdIFFT(const GPUMatrix<TType>* in_mat,
                 GPUMatrix<TType>* out_mat,
                 unsigned int num_z)
```

- Calculates the centered fast Fourier transform of the input matrix with defined size and pattern.

```
void CenterdFFTpattern(const GPUMatrix<TType>* in_mat,
                       GPUMatrix<TType>* out_mat,
                       unsigned int num_z)
```

- Calculates the centered inverse fast Fourier transform of the input matrix with defined size and pattern.

```
void CenterdIFFTpattern(const GPUMatrix<TType>* in_mat,
                        GPUMatrix<TType>* out_mat,
                        unsigned int num_z)
```

- Applies the calculated high frequency penalty (see section 1.1.4) to the input matrix (equation 1.32).

```
oid ApplyW(const GPUMatrix<TType>* in_mat,
           GPUMatrix<TType>* out_mat,
           unsigned int num_z)
```

- Applies the calculated high frequency penalty (see section 1.1.4) to the input matrix (equation 1.33).

```
void ApplyWH(const GPUMatrix<TType>* in_mat,
             GPUMatrix<TType>* out_mat,
             unsigned int num_z)
```

- Calculates the differential operater DF^H described in equation 1.35.

```
void ApplyDFH(GPUMatrix<TType>* rhs_mat,
              const GPUMatrix<TType>* dx)
```

- Calculates the differential operater $DF^H DF$ described in equation 1.34 and 1.35.

```
void ApplyM(GPUMatrix<TType>* gu,
            GPUMatrix<TType>* gc,
            const GPUMatrix<TType>* du,
            const GPUMatrix<TType>* dc)
```

- Copy the defined number of matrices num_z from matrix *in_mat* to *out_mat*

```
void CopyMatrixZ(const GPUMatrix<TType>* in_mat,
                 GPUMatrix<TType>* out_mat,
                 unsigned int num_z)
```

- Calculates the fixed Lipschitz constant as shown in section 1.1.9 and returns the value.

```
TType2 calcLipschitz()
```

- Pure virtual declaration of the method *Solve*.

```
virtual void Solve(const GPUMatrix<TType>* u,
                   const GPUMatrix<TType>* c,
                   const GPUMatrix<TType>* rhs,
                   const GPUMatrix<TType>* u0,
```

```
5                       unsigned maxits, TType2 alpha,
6                       TType2 beta,
7                       GPUMatrix<TType>* du,
8                       GPUMatrix<TType>* dc) = 0;
```

Private methods

- Initializes the class member variables.

```
1 void Init()
```

- randomcalc returns a random value between 0 and 1 with the defined value type TType.
  The argument 'i' is the defined seed value.

```
1 typename TType::value_type randomcalc(int i)
```

- Test method to verify the given IRGN parameters.
  The return value is true if the configuration is valid.

```
1 bool irgn_param_test(IRGN_Params &param)
```

2.3.4 L2Solve class

Figure 2.10 and 2.11 show the declaration structure of the L2Solve class.

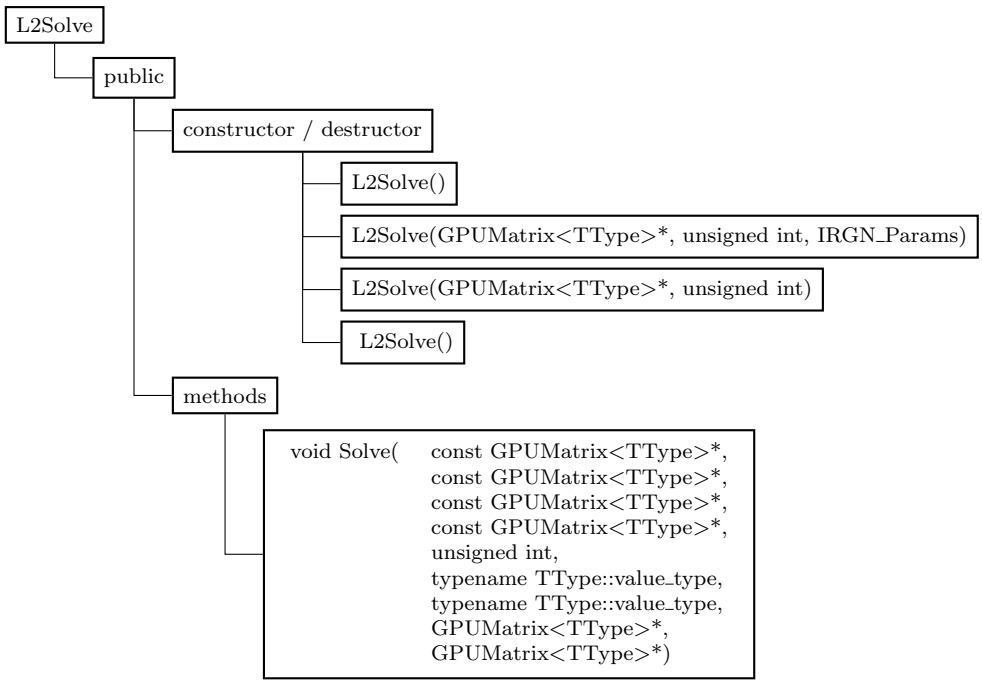


Figure 2.10: L2Solve public Class structure

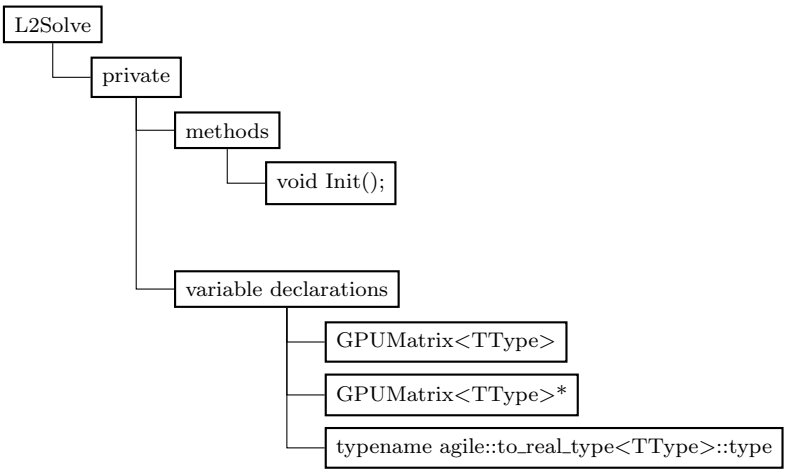


Figure 2.11: L2Solve private Class structure

Description of methods

The L2Solve class implements the algorithm with L2-regularization by overriding the pure virtual solve-method located in IRGN basis class.

- The default constructor creates an empty L2Solve object.

```
L2Solve()
```

- This constructor initializes an L2Solve object with defined data and configuration parameters.

```
L2Solve( GPUMatrix<TType>* coil,
         unsigned int num_coils,
         IRGN_Params param )
```

- Constructor to initialize an L2Solve object with defined data.

```
L2Solve( GPUMatrix<TType>* coil,
         unsigned int num_coils )
```

- Destructor frees used resources.

```
~L2Solve()
```

- The Solve method calculates the update values per IRGN step with L2 regularization.

```
void void Solve(const GPUMatrix<TType>* u,
                const GPUMatrix<TType>* c,
                const GPUMatrix<TType>* rhs,
                const GPUMatrix<TType>* u0,
                unsigned maxits,
                typename TType::value\_type alpha,
                typename TType::value\_type beta,
                GPUMatrix<TType>* du,
                GPUMatrix<TType>* dc);
```

- Initializes the class member variables.

```
void Init()
```

2.3.5 TVSolve class

Figure 2.12 and 2.13 show the declaration structure of the TVSolve class.

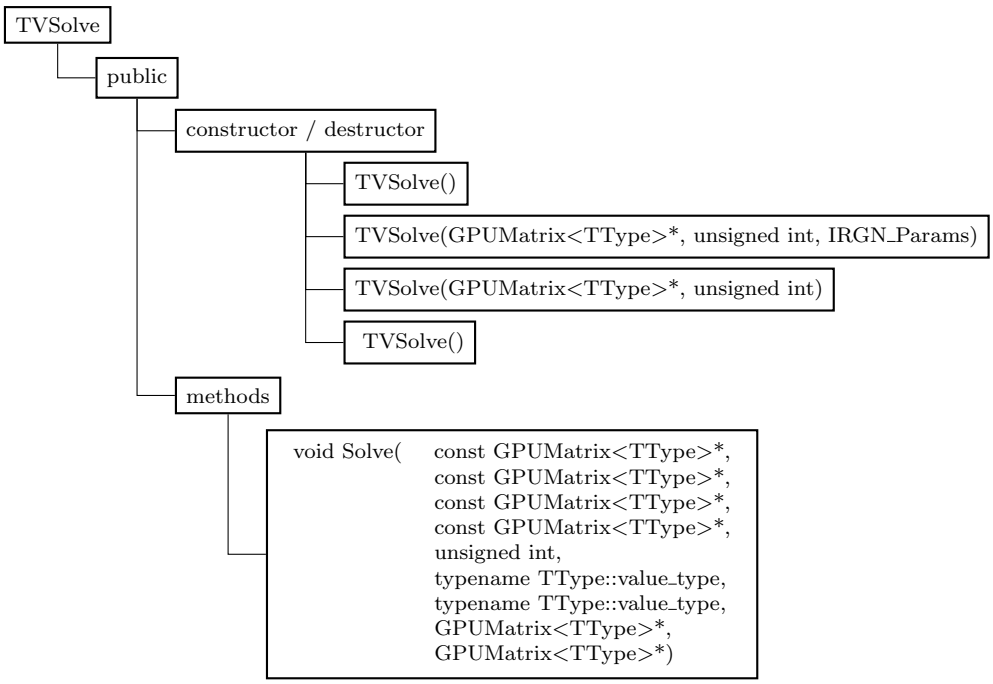


Figure 2.12: TVSolve public Class structure

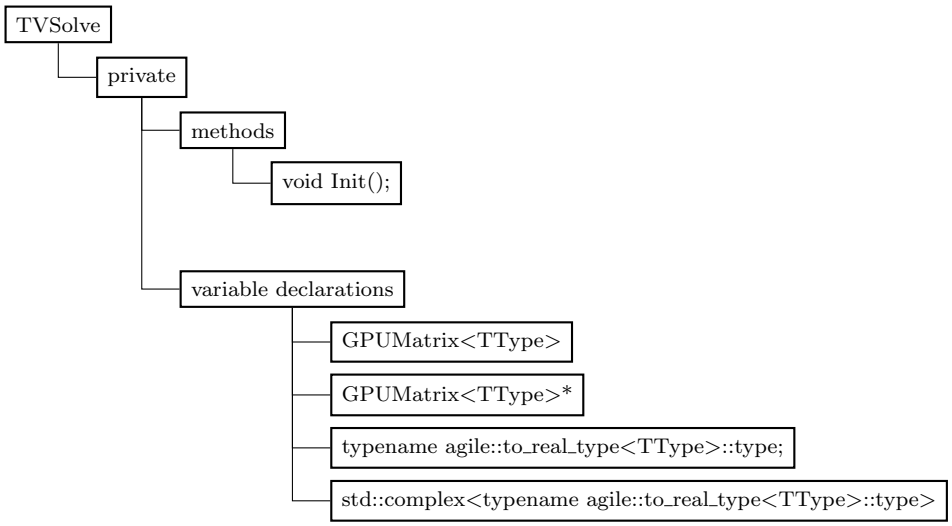


Figure 2.13: TVSolve private Class structure

Description of methods

The TVSolve class implements the algorithm with TV-regularization by overriding the pure virtual solve-method located in IRGN basis class.

- The default constructor creates an empty TVSolve object.

```
1 TVSolve()
```

- This constructor initializes an TVSolve object with defined data and configuration parameters.

```
1 TVSolve( GPUMatrix<TType>* coil,
2          unsigned int num_coils,
3          IRGN_Params param )
```

- Constructor to initialize an TVSolve object with defined data.

```
1 TVSolve( GPUMatrix<TType>* coil,
2          unsigned int num_coils )
```

- Destructor frees used resources.

```
1 ~TVSolve()
```

- The Solve method calculates the update values per IRGN step with TV regularization.

```
void void Solve(const GPUMatrix<TType>* u,
                const GPUMatrix<TType>* c,
                const GPUMatrix<TType>* rhs,
                const GPUMatrix<TType>* u0,
                unsigned maxits,
                typename TType::value\_type alpha,
                typename TType::value\_type beta,
                GPUMatrix<TType>* du,
                GPUMatrix<TType>* dc);
```

- Initializes the class member variables.

```
void Init();
```

2.3.6 TGVSolve class

Figure 2.14 and 2.15 show the declaration structure of the TVSolve class.

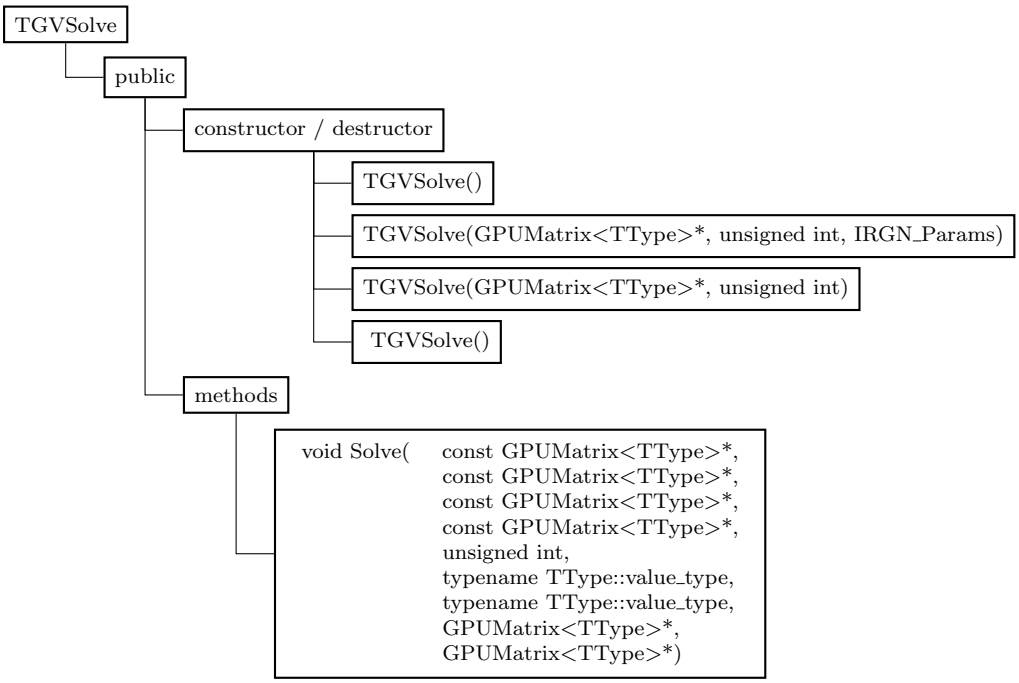


Figure 2.14: TGVSolve public Class structure

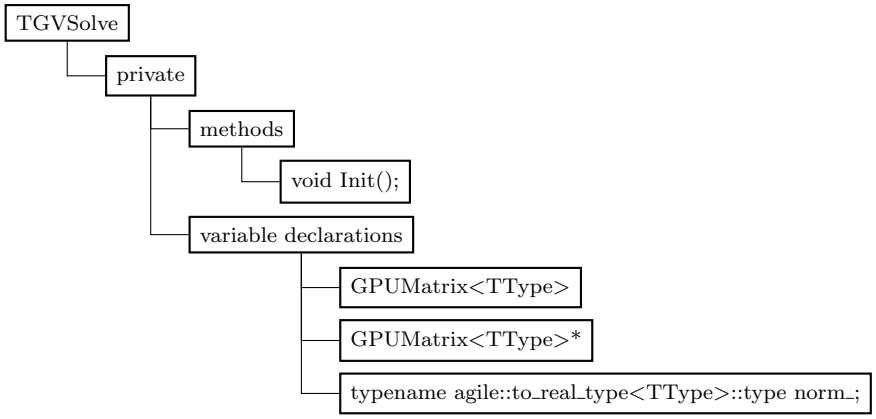


Figure 2.15: TGVSolve private Class structure

Description of methods

The TGVSolve class implements the algorithm with TV-regularization by overriding the pure virtual solve-method located in IRGN basis class.

- The default constructor creates an empty TGVSolve object.

```
TGVSolve()
```

- This constructor initializes an TGVSolve object with defined data and configuration parameters.

```
TGVSolve( GPUMatrix<TType>* coil,
          unsigned int num_coils,
          IRGN_Params param )
```

- Constructor to initialize an TGVSolve object with defined data.

```
TGVSolve( GPUMatrix<TType>* coil,
          unsigned int num_coils )
```

- Destructor frees used resources.

```
~TGVSolve()
```

- The Solve method calculates the update values per IRGN step with TGV regularization.

```
void void Solve(const GPUMatrix<TType>* u,
                const GPUMatrix<TType>* c,
                const GPUMatrix<TType>* rhs,
                const GPUMatrix<TType>* u0,
                unsigned maxits,
                typename TType::value\_type alpha,
                typename TType::value\_type beta,
                GPUMatrix<TType>* du,
                GPUMatrix<TType>* dc);
```

- Initializes the class member variables.

```
void Init();
```

2.4 Useful functions

2.4.1 GPU timer

The GPUTimer Class is integrated within the Agile library.
The public method *stop()* returns the elapsed time computed by the wrapped CUDA function *cudaEventElapsedTime*.
The resulting value is in milliseconds with a resolution of approximately 0.5 microseconds.

- Possible method call structure.

```
agile::GPUTimer gpuTimer;
gpuTimer.start();
double timerValueInS = gpuTimer.stop() * 0.001;
```

2.4.2 matrixhelper.h - Logger

The matrixhelper.h file includes global console output helper functions for logging purposes.
The functionality can be divided into the categories console logger and file logger.

Console logger

- Prints a vector 'x' to the console with a description string 'string'.

```
template <typename TType>
void output(const char* string,
            const std::vector<TType>& x)
```

- Prints a matrix 'data' with defined number of rows and columns to the console with a description string 'string'.

```
template <typename TType>
void output(const char* string,
            unsigned num_rows,
            unsigned num_columns,
            const std::vector<TType>& data)
```

File logger

- Initializer for the matrix logger with defined filename.

```
void init_matrixlog(std::fstream &myfile,
                    const char * file_name);
```

- End the matrix logger to close the file-stream and provide the logged file.

```
void close_matrixlog(std::fstream &myfile);
```

- Matrix-logger for a vector or matrix according to the given number of rows and columns.

```
template <typename TType>
void matrixlog( std::fstream &myfile,
                const char* string,
                unsigned num_rows,
                unsigned num_columns,
                const std::vector<TType>& data);
```

- Matrix-logger for single value.

```
template <typename TType>
void matrixlog( std::fstream &myfile,
                const char* string,
                const TType data);
```

- Matrix-Logger for string. A reference to the log-file and the description string 'string' need to be provided.

```
void matrixlog( std::fstream &myfile,
                const char* string);
```

The following example shows a possible function call structure:

```
std::fstream myfile;
init_matrixlog(myfile,"matrixlog.txt")
matrixlog(myfile,"Matrix 1 "); // "Matrix 1 " into matrixlog.txt
close_matrixlog(myfile);
```

2.4.3 File input / output functions for Matlab

The Agile library also provides read and write functions for C++ and MATLAB® located at include/agile/io. The file *file.hpp* implements the extern functions

readMatrixFile3D and *writeMatrixFile3D*, which can handle multi-coil image raw data.

- Reads the 3d matrix data and dimensions from a given file.

```
bool readMatrixFile3D(const char* file_name,
                      unsigned& num_rows,
                      unsigned& num_columns,
                      unsigned& num_coils,
                      std::vector<std::complex<TValueType> >&
    data);
```

- Writes the 3d matrix data with given dimensions into a file.

```
bool writeMatrixFile3D(const char* file_name,
                       unsigned num_rows,
                       unsigned num_columns,
                       unsigned num_coils,
                       std::vector<std::complex<Type> >& data)
```

Matlab

The files writeMatlab2Bin3D.m and readMatlab2Bin3D.m (include/agile/io) can read and write the defined file structure (see section 2.4.3 File structure) for mulit-coil raw data within the MATLAB® framework.

- Writes the 3d matrix data into a file.

```
function writeMatlab2Bin3D(matrix, filename)
```

- Reads the 3d matrix data and dimensions from a given file.

```
function matrix = readBin2Matlab3D( filename,
                                    numRows,
                                    numColumns,
                                    numCoils,
                                    numBytesPerEntry )
```

File structure

The binary written or read file has the following data structure:

declaration	info
unsigned num_rows	number of matrix rows
unsigned num_columns	number of matrix columns
unsigned num_coils	number of matrix coils
unsigned num_bytes_per_entry	number of matrix rows
bool is_complex	true if the stored entries are complex values
List<Type> data	stored matrix data with template defined type

Table 2.1: Declaration of the matrix file format

num_rows, *num_columns* and *num_coils* define the size of the matrix.

2.5 DCMTK library

DCMTK is a collection of libraries and applications implementing large parts of the Digital Imaging and Communications in Medicine (DICOM) standard. It includes software for examination, construction and conversion of DICOM image files, handling offline media, sending and receiving images over a network connection, as well as demonstrative image storage and worklist servers. DCMTK is written in a mixture of ANSI C and C++.
The Dicom Toolkit (DCMTK) Library can be downloaded from their official homepage located at `http://dicom.offis.de/dcmtk.php.en`, but it is also available in the package manager. If the library is downloaded and installed manually, one can choose the installation-paths freely.
By using the YaST package manager the standard-paths (/usr/local/bin and /usr/local/include) for installation are used. The used version is 3.6.0.

2.6 CMake

The CMake package can be downloaded from their homepage https://cmake.org or installed via the package manager. The used version is 2.8.10.2.

2.6.1 ccmake

ccmake is a commandline editor to configure the cmake project environment. It is part of the cmake package.

2.6.2 CMakeLists.txt

CMakeLists.txt file contains a set of definitions and instructions describing the project's source files and targets (executable, library or both).
The IRGN project CMakeLists configuration is located in the AGILE library (AGILE/apps/imt_irgn/CMakeLists.txt).

- Add include directories to the build.

```
# source definitions
INCLUDE_DIRECTORIES( ${CMAKE_SOURCE_DIR}
                     "${IMT_IRGN_SOURCE_DIR}/gui"
                     "${IMT_IRGN_BINARY_DIR}/gui"
                     "${IMT_IRGN_SOURCE_DIR}"
                     "${IMT_IRGN_BINARY_DIR}")

# include definitions
INCLUDE_DIRECTORIES( "${AGILE_SOURCE_DIR}/include"
                     "${AGILE_BINARY_DIR}/include"
                   )
```

- Link instruction for the agile library.

```
link_libraries(agile)
```

- Define the source files used in the project.

```
SET(IMT_IRGN_SRCS_CXX
    imt_mrcmd.cpp
    main.cpp
    qt-source-files)
```

- Add the source definitions to the executable.

```
ADD_EXECUTABLE(IMT_IRGN ${IMT_IRGN_SRCS_CXX})
```

- To link the application following command is needed.

```
TARGET_LINK_LIBRARIES(IMT_IRGN ${QT_LIBRARIES})
```

- To add the cuda and cufft library to the IMT_IRGN build the following call needs to be added.

```
cuda_add_cufft_to_target (IMT_IRGN)
```

- As an additional post-build step, the resulting executable is copied to a defined path.

```
add_custom_command(TARGET IMT_IRGN
                   POST_BUILD
                   COMMAND ${CMAKE_COMMAND} -E copy $<
  TARGET_FILE:IMT_IRGN> "${AGILE_SOURCE_DIR}/bin")
```

2.7 gcc-C++

For developing with Qt and the CUDA toolchain the gcc-c++ library is needed. If not already installed it can be be added via a package manager. The used version is 4.7.2.

3 Results

3.1 Calculation and precision analysis

To evaluate the IRGN algorithm calculation results, the reconstructed image is calculated with equation 3.5, which represents the magnitude of the calculated spin densities.
The pseudorandom brain data set included in the Matlab implementation package of the IRGN algorithm was used to calculate the results (*Clason, Bredies* 2011). It consists of a 12 channel brain data acquisition compressed to 6 virtual channels with a pseudorandom sampling pattern (Knoll, 2011).

All calculations in Matlab are performed with double precision (*Double-precision floating-point format* 2018). The results gained from the CUDA implementations are calculated in single and double precision and compared to the Matlab implementation results.
A comparison between the single (float) and double precision floating point format is shown in table 3.1.

type	minimum	maximum	epsilon
float	1.17549e-38	3.40282e+38	1.19209e-07
double	2.22507e-308	1.79769e+308	2.22045e-16

Table 3.1: Minimum, maximum and epsilon value comparison between float and double precision

Table 3.2 shows the calculation differences between the CUDA and Matlab implementation.

	$\overline{I_{rel}}$	$\overline{I_{abs}}$	$Matlab_{max}$	$CUDA_{max}$	$Matlab_{min}$	$CUDA_{min}$
Inv	1.82e-08	3.66e-14	3,12e-06	3,12e-06	6,60e-07	6,60e-07
L2	1.09e-03	2.20e+02	3,70e+05	3,71e+05	3,71e+04	3,70e+04
TV	7.09e-03	1.45e+03	3,40e+05	3,40e+05	6,33e+04	6,70e+04
TGV	6.00e-03	1.23e+03	3,44e+05	3,43e+05	6,33e+04	6,23e+04

Table 3.2: Matlab and CUDA-powered IRGN calculated image differences
$\overline{I_{rel}}$: Mean of the relative pixel-intensity differences
$\overline{I_{abs}}$: Mean of the absolute pixel-intensity differences
$Matlab_{max}$: Maximum Matlab calculated intensity value
GPU_{max}: Maximum GPU calculated intensity value
$Matlab_{min}$: Minimum Matlab calculated intensity value
GPU_{min}: Minimum GPU calculated intensity value

The mean of the relative pixel-intensity differences $\overline{I_{rel}}$ between the CUDA and Matlab reconstruction results is calculated with equation 3.1.
Equation 3.2 calculates the mean of the absolute error $\overline{I_{abs}}$.

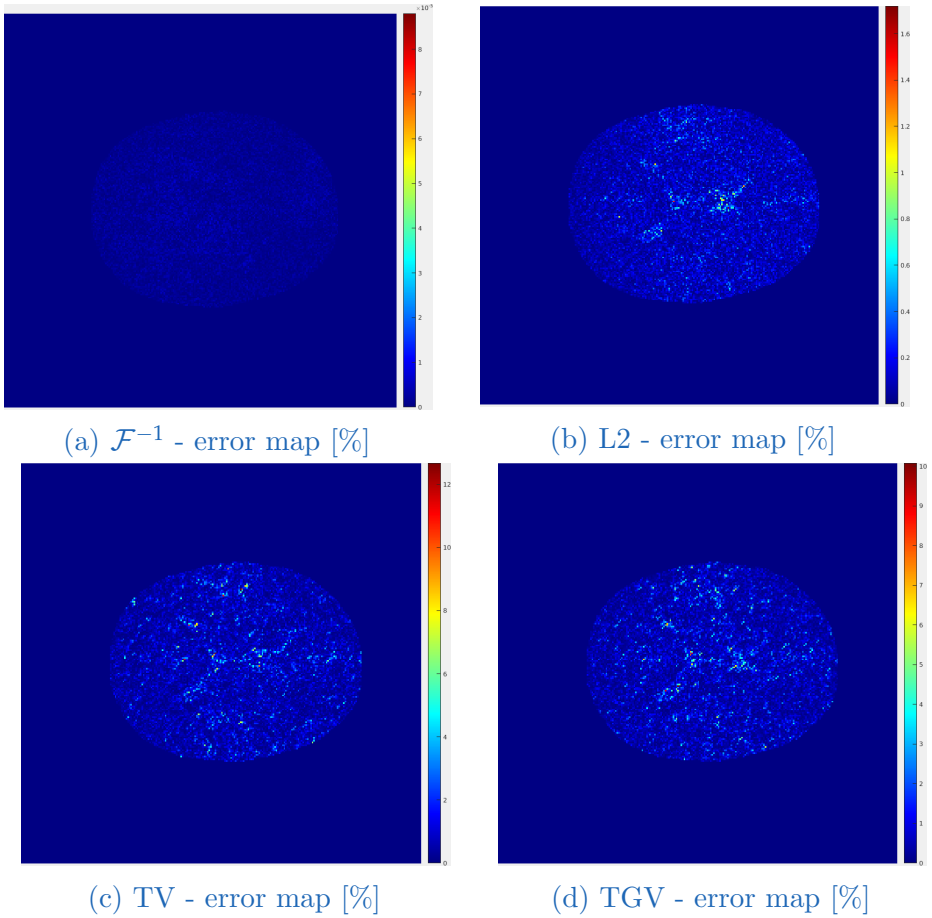
The presented results for the relative and absolute errors are calculated within the region of interest (ROI) in the image, which is the region of the object without the skull structure.

$$\overline{I_{rel}} = mean(\frac{\mid I_{CUDA} - I_{Matlab} \mid}{I_{Matlab}}) \tag{3.1}$$

$$\overline{I_{abs}} = mean(\mid I_{CUDA} - I_{Matlab} \mid) \tag{3.2}$$

Figure 3.1 shows the error maps, in percent, for the different reconstruction results. To calculate the error maps, the relative error per pixel is calculated with equation 3.3.

$$I_{rel}[\%] = 100 * \frac{\mid I_{CUDA} - I_{Matlab} \mid}{I_{Matlab}} \tag{3.3}$$



(a) $\mathcal{F}^{-1}$ - error map [%]

(b) L2 - error map [%]

(c) TV - error map [%]

(d) TGV - error map [%]

Figure 3.1: $\overline{I_{diff}}$[%] error maps for the defined reconstruction method ($\mathcal{F}^{-1}$, L2-,TV-,TGV-regularization)

To analyze the resulting differences (equation 3.4) between CUDA and Matlab the defined methods and calculation-parts are unit-tested with random generated moc-data.
The results for a calculation precision in double and single, are shown in table 3.3.
The unit-tests for L2Solve, TVSolve and TGVSolve do not include the calculation of the differential of the function F(x) (method call *ApplyM*). The Results are are shown for different numbers of subgradient steps.

$$\overline{diff} = mean(||result_{CUDA}| - |result_{Matlab}||) \tag{3.4}$$

	$\overline{diff_{double}}$	$\overline{diff_{float}}$
Scale	0	0
ApplyM	1.3e-11	1.2e-11
ApplyDFH	1.7e-11	1.7e-11
Normalize	4.9e-09	4.9e-09
ApplyWH	6.1e-11	6.1e-11
ApplyW	3.4e-10	3.3e-10
Residual	1.3e-11	2.0e-11
$L2Solve_{2iter}$	1.1e-18	1.1e-18
$TVSolve_{2iter}$	4.2e-17	4.2e-17
$TGVSolve_{2iter}$	4.3e-17	4.3e-17
$L2Solve_{100iter}$	0	0
$TVSolve_{100iter}$	2.1e-14	1.6e-14
$TGVSolve_{100iter}$	1.0e-13	4.7e-14
$L2Solve_{640iter}$	0	0
$TVSolve_{640iter}$	1.5e-13	4.2e-13
$TGVSolve_{640iter}$	5.1e-13	6.9e-13
Postprocess	1.5e-09	1.5e-09

Table 3.3: The mean differences (equation 3.1) between the CUDA and Matlab defined methods and calculation-parts.

Table 3.4 shows the achieved GPU and CPU reconstruction times.
The CPU Matlab calculations are performed on a Intel Xeon E7-4830 v4 2.00GHz.
The Nvidia Tesla K40c is used for all CUDA calculations.

The GPU utilization factor in % for single precision is approximately 45% for IRGN-L2, TV and TGV.
For double precision the utilization factor for IRGN-L2 and TV is 50% and for IRGN-TGV 75%.

The GPU memory usage for single precision calculation is approximately

180MB.
For double precision the global memory usage rises up to 230MB.

method	CPU [s]	GPU_{double} [s]	GPU_{float} [s]	$speedup_{double}$	$speedup_{float}$
L2	43	14	12	3	3.3
TV	46	14	12	3.3	3.8
TGV	51	15	5	3.4	10

Table 3.4: Comparison of the reconstruction times between CPU and GPU for the different regularization methods (L2, TV, TGV) and precision

Table 3.5 shows the configuration of the IRGN algorithm used for all reconstruction results.

parameter	value	info
maxit	5	maximum number of IRGN iterations
β_{min}	0	final value of beta: 0: no TGV effect, >0 effect
α_0	1	initial penalty α_0 (L2, sensitivites)
β_0	1	initial penalty β_0 (image)
α_{min}	0	final value of α
α_q	1/10	reduction factor for α
β_q	1/5	reduction factor for β
tvits	20	initial number of gradient steps
tv_{max}	1000	upper bound on number of gradient steps
ρ_0	0	initial values of the proton density

Table 3.5: Configuration of the IRGN algorithm

The tables 3.6, 3.7 and 3.8 show the difference between the CUDA and Matlab residual-norm r_{norm} (equation 1.12) per IRGN step n.

n	gradient steps	α	β	CUDA r_{norm}	Matlab r_{norm}
1	20	1	1	100	100
2	40	0.1	0.2	99.79	99.79
3	80	0.01	0.04	92.67	92.59
4	160	0.001	8e-3	51.33	53.05
5	320	1e-4	16e-4	5.76	5.70

Table 3.6: Residual norm and regularization factors per gauss-newton step for IRGN-L2 calculation

n	gradient steps	α	β	CUDA r_{norm}	Matlab r_{norm}
1	20	1	1	100	100
2	40	0.1	0.2	65.96	64.378
3	80	0.01	0.04	33.14	33.167
4	160	0.001	8e-3	16.45	16.59
5	320	1e-4	16e-4	8.62	8.675

Table 3.7: Residual norm and regularization factors per gauss-newton step for IRGN-TV calculation

n	gradient steps	α	β	CUDA r_{norm}	Matlab r_{norm}
1	20	1	1	100	100
2	40	0.1	0.2	64.93	64.93
3	80	0.01	0.04	29.25	29.24
4	160	0.001	8e-3	16.83	15.06
5	320	1e-4	16e-4	8.43	8.58

Table 3.8: Residual norm and regularization factors per gauss-newton step for IRGN-TGV calculation

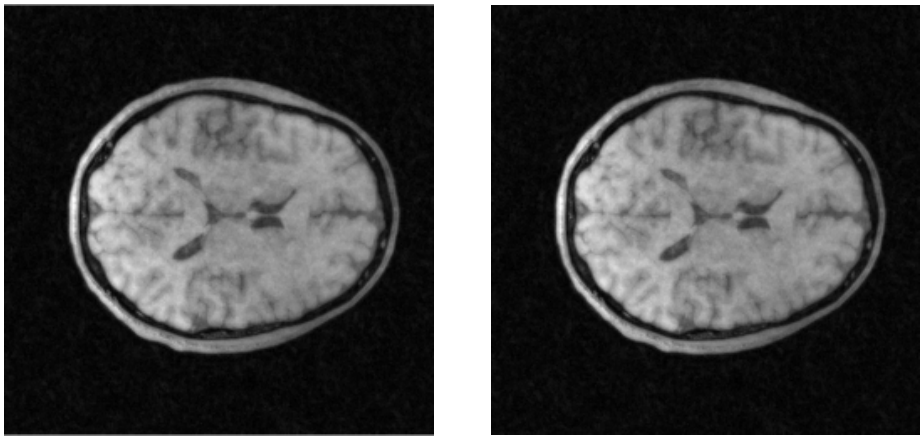
3.2 IRGN reconstruction results

Using equation 3.5 to reconstruct the under-sampled rawdata y, without using the IRGN algorithm, results in figure 3.2.

$$I_{\mathcal{F}^{-1}} = \sqrt{\sum_{i=1}^{N} |\mathcal{F}^{-1}(y_i)|^2} \tag{3.5}$$

Image reconstruction results, calculated with the IRGN algorithm, are computed with equation 3.6.

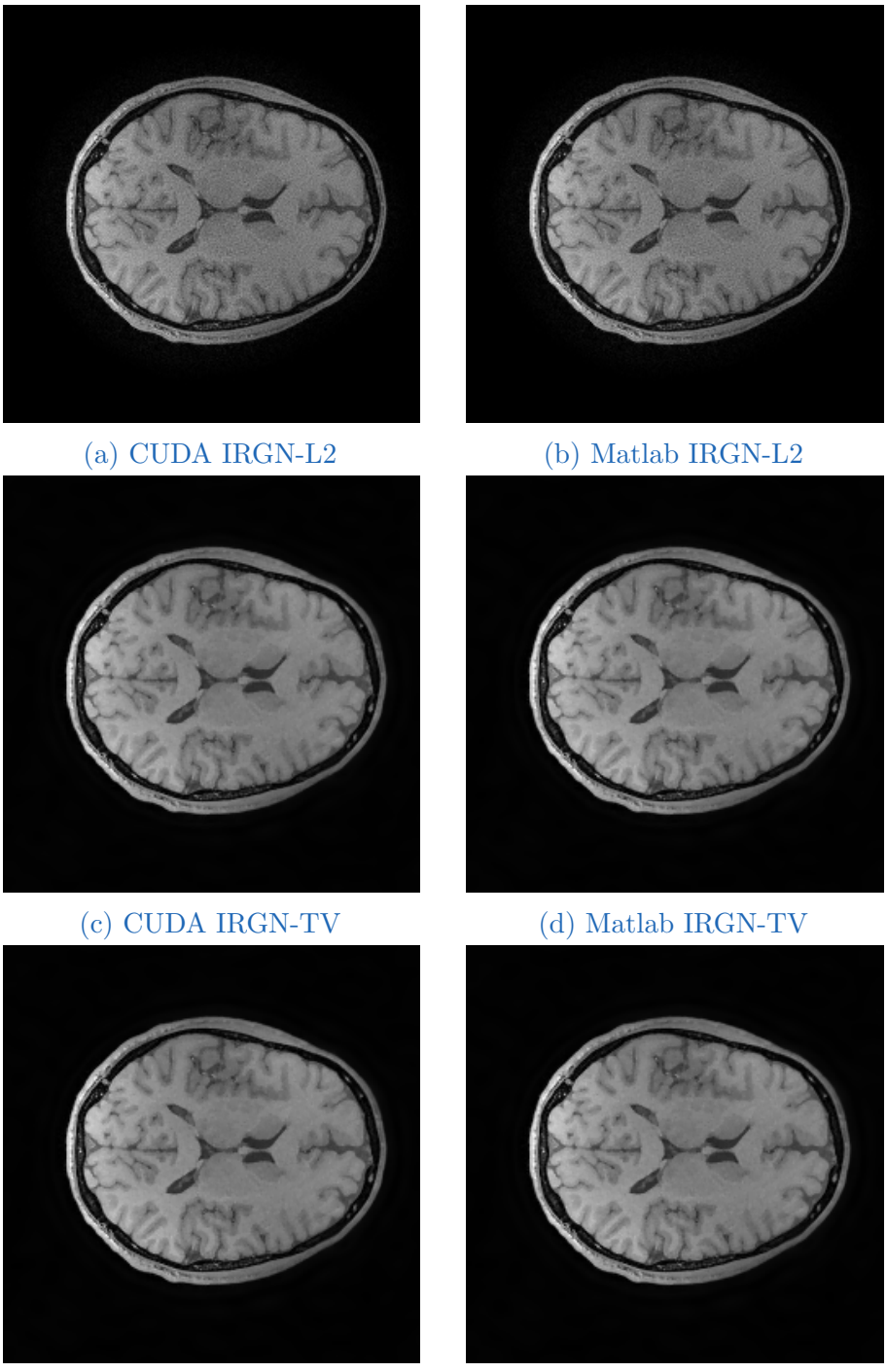
$$I_{irgn} = |\rho_{irgn}| \tag{3.6}$$



(a) CUDA $\mathcal{F}^{-1}$ (b) Matlab $\mathcal{F}^{-1}$

Figure 3.2: Inverse Fourier transformed reconstruction for the random sampled brain data

Figure 3.3 shows the reconstructed image with different regularization terms for the CUDA and Matlab implementation.

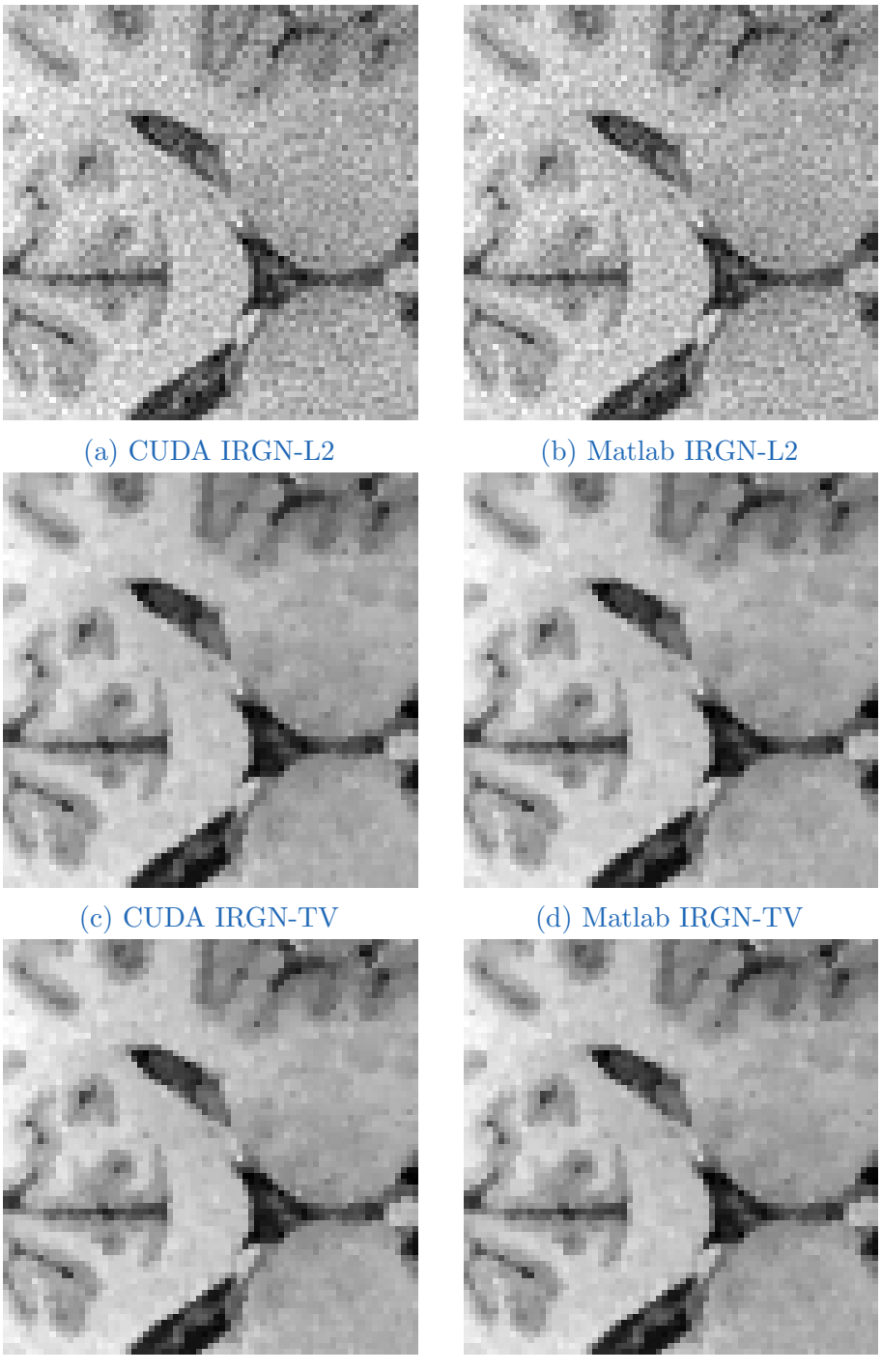


(a) CUDA IRGN-L2 (b) Matlab IRGN-L2

(c) CUDA IRGN-TV (d) Matlab IRGN-TV

(e) CUDA IRGN-TGV (f) Matlab IRGN-TGV

Figure 3.3: IRGN reconstruction results with different regularizations for the CUDA and Matlab implementation



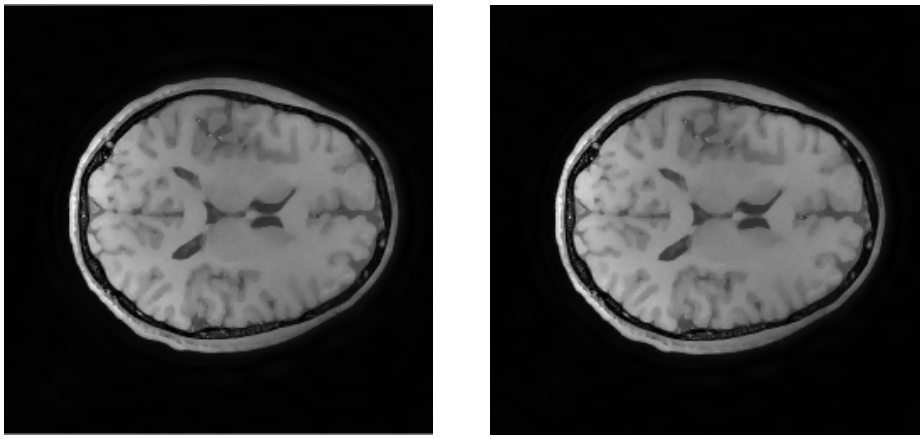
(a) CUDA IRGN-L2 (b) Matlab IRGN-L2

(c) CUDA IRGN-TV (d) Matlab IRGN-TV

(e) CUDA IRGN-TGV (f) Matlab IRGN-TGV

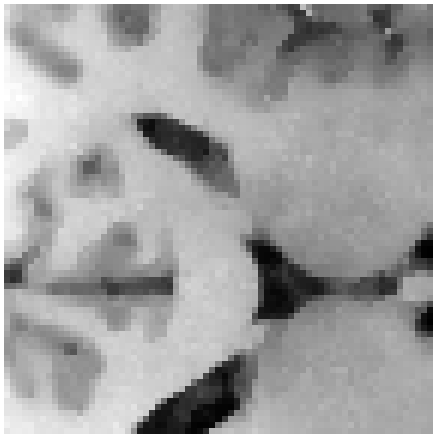
Figure 3.4: Close-up view of the IRGN reconstruction results with different regularizations for the CUDA and Matlab implementation

For a more pronounced TV and TGV regularization effect, the configuration parameter $beta_{min}$ is changed to a larger value of 1e-2 for TV and 5e-3 for TGV regularization.
The resulting images for IRGN-TV and IRGN-TGV are shown in figure 3.5 and 3.6.

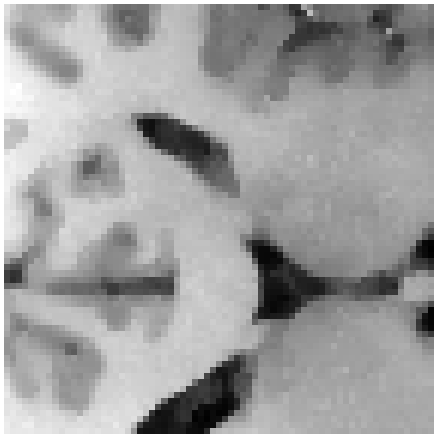


(a) CUDA IRGN-TV

(b) Matlab IRGN-TV

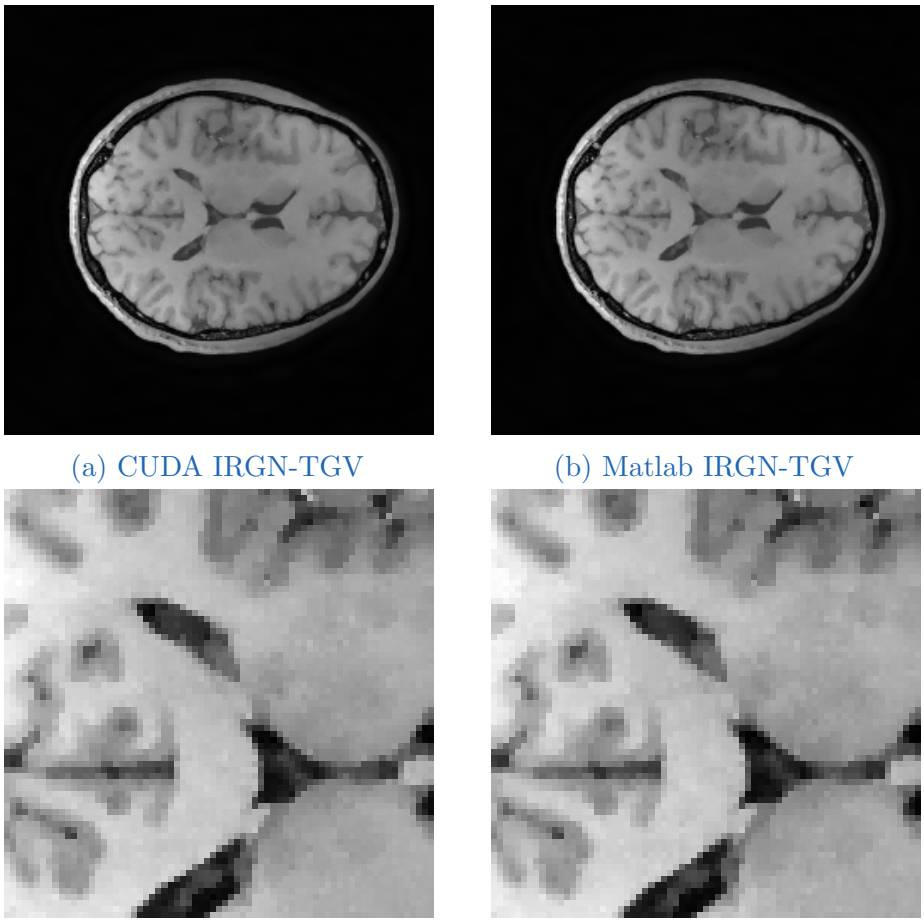


(c) CUDA IRGN-TV



(d) Matlab IRGN-TV

Figure 3.5: IRGN reconstruction results with TV regularization and a $beta_{min}$ value equal to 1e-2
Closeup-views are shown in figure 3.5c and 3.5d



(a) CUDA IRGN-TGV (b) Matlab IRGN-TGV

(c) CUDA IRGN-TGV (d) Matlab IRGN-TGV

Figure 3.6: IRGN reconstruction results with TGV regularization and a $beta_{min}$ value equal to 5e-3
Closeup-views are shown in figure 3.6c and 3.6d

4 Discussion

The CUDA powered IRGN algorithm performs up to a factor of 10 times faster than the Matlab implementation, which is performed on the CPU (table 3.4). The different utilization factors are reported by the nvidia-smi commandline tool.
For a given time period, the utilization factor represents what percentage of time one or more GPU kernels are active.
During an inactive state, between every kernel execution the GPU handles the memory management and the instruction calls.
To gain a high utilization factor, the whole IRGN execution takes place on the GPU without any device to host and vice versa communication during the calculation process.
Accoring to Amdahl's Law (figure 1.7) a speedup factor of 10 means that the IRGN CUDA implementation is 90% parallelized.
Because of the consistent implementation of the IRGN regularization methods, the origin for the different speedup factors lies on the compiler and architecture optimization depending on code complexity and precision-type.

Values of 230MB for double precision and 180MB for single precision, reveal a low global memory usage by the CUDA powered IRGN application.

The calculated differences of the image intensity values $\overline{I_{rel}}$ and $\overline{I_{abs}}$ shown in table 3.2, indicate that the error increases with the complexity of the calculation.
Table 3.6 also shows different calculated residual norm r_{norm} values between the CUDA and Matlab implementation.
Because of the error propagation per iteration step, which for example comes from the calculation of different finite differences, the reconstruction results between CUDA and Matlab slightly differ.
The error maps shown in figure (3.1) indicate that there are no systematical errors but differences primarily at the tissue edges.

In addition to the analyzed pixel values, all methods and calculation-parts of the IRGN algorithm are unit-tested for double and single precision (see table 3.3).
Due to floating-point rounding errors, the mean differences in the results between CUDA and Matlab occur (table 3.3), but lie in a range lower than the epsilon value of 1.19e-07 for single precision.

The images (with a bit-depth of 8bit) in figure 3.3 and the close-up views in figure 3.4 show visually an identical result for the CUDA and Matlab reconstructions.
It is also shown that the IRGN algorithm with TV and TGV regularization clearly outperforms the approach with L2 regularization.
A higher regularization factor β for the variational approaches, leads to a better denoised reconstruction result (see figure 3.5 and 3.6).
Due to the good signal to noise ratio (SNR) of the test data, only a low amount of TV regularization is applied and therefore the reconstruction results do not show any staircasing artifacts.

5 Conclusion

Executing the algorithm with the CUDA powered IRGN implementation gives slightly different numerical results compared with the Matlab implementation. Next steps to find the origin of the calculation differences between GPU and CPU would be an extended unit-testing, PTX debugging, software and hardware updates.

In general it can be said, that the achieved acceleration of the computation times can benefit the processing of large data-sets in daily clinical practice.

Bibliography

Amdahl's law (2018). In: *Wikipedia*. Page Version ID: 854098361. URL: https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=854098361 (visited on 08/13/2018) (cit. on p. 20).

Bernstein et al. (2004). In: *Handbook of MRI Pulse Sequences*. Academic Press (cit. on p. 1).

BFGS algorithm (1970). In: *Wikipedia*. Page Version ID: 880084852. URL: https://en.wikipedia.org/w/index.php?title=Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm&oldid=880084852 (visited on 01/25/2019) (cit. on p. 3).

BLAS (Basic Linear Algebra Subprograms) (2018). URL: http://www.netlib.org/blas/ (visited on 08/19/2018) (cit. on p. 25).

Bredies, K., K. Kunisch, and T. Pock (2010). "Total Generalized Variation." In: *SIAM Journal on Imaging Sciences* 3.3, pp. 492–526. DOI: 10.1137/090769521. URL: https://doi.org/10.1137/090769521 (visited on 01/06/2019).

Clason, Bredies (2011). URL: https://www.tugraz.at/institute/imt/research/mr-image-reconstruction/ (visited on 2011) (cit. on pp. 39, 62).

CUDA C Programming Guide (2018). URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 08/15/2018) (cit. on pp. 22, 24).

cuFFT :: CUDA Toolkit Documentation (2018). URL: https://docs.nvidia.com/cuda/cufft/index.html#introduction (visited on 08/21/2018) (cit. on p. 26).

Deshmane, Anagha et al. (2012). "Parallel MR imaging." In: *Journal of Magnetic Resonance Imaging* 36.1, pp. 55–72. ISSN: 1522-2586. DOI: 10.1002/jmri.23639. URL: http://dx.doi.org/10.1002/jmri.23639.

Double-precision floating-point format (2018). In: *Wikipedia*. Page Version ID: 871226723. (Visited on 12/15/2018) (cit. on p. 62).

Finite difference (2018). In: *Wikipedia*. URL: https://en.wikipedia.org/wiki/Finite_difference (visited on 12/03/2018) (cit. on p. 13).

Flynn's taxonomy (2018). In: *Wikipedia*. Page Version ID: 850899167. URL: https://en.wikipedia.org/w/index.php?title=Flynn%27s_taxonomy&oldid=850899167 (visited on 08/15/2018) (cit. on p. 19).

Fréchet derivative (2018). In: *Wikipedia*. Page Version ID: 850209316. URL: https://en.wikipedia.org/w/index.php?title=Fr%C3%A9chet_derivative&oldid=850209316 (visited on 08/04/2018).

Freiberger, Manuel et al. (2013). "The AGILE library for image reconstruction in biomedical sciences using graphics card hardware acceleration." In: *Computing in Science and Engineering* 15, pp. 34–44 (cit. on p. 26).

Haacke et al. (1999). In: *Magnetic Resonance Imaging: Physical Principles and Sequence Design*. John Wiley and Sons (cit. on p. 2).

Hager et al. (2006). In: *A survey of nonlinear conjugate gradient methods*. This paper reviews the development of different versions of nonlinear conjugate gradient methods, with special attention given to global convergence properties. (cit. on p. 3).

Kimpe, Tom and Tom Tuytschaever (2007). "Increasing the number of gray shades in medical display systems–how much is enough?" In: *Journal of digital imaging* 20.4, pp. 422–432. ISSN: 0897-1889. JSTOR: 17195900. URL: https://www.ncbi.nlm.nih.gov/pubmed/17195900.

Knoll, Florian (2011). "Constrained MR Image Reconstruction of Undersampled Data from Multiple Coils." In: (cit. on pp. 4, 14, 17, 18, 62).

Knoll, Florian, Kristian Bredies, et al. (2011). "Second order total generalized variation (TGV) for MRI." In: *Magnetic resonance in medicine* 65.2, pp. 480–491. ISSN: 1522-2594. JSTOR: 21264937. URL: https://www.ncbi.nlm.nih.gov/pubmed/21264937 (cit. on p. 12).

Knoll, Florian, Christian Clason, et al. (2012). "Parallel imaging with nonlinear reconstruction using variational penalties." In: *Magnetic Resonance in Medicine* 67.1, pp. 34–41. ISSN: 1522-2594. DOI: 10.1002/mrm.22964. URL: http://dx.doi.org/10.1002/mrm.22964.

Levenberg–Marquardt algorithm (2018). In: *Wikipedia*. Page Version ID: 852940243. URL: https://en.wikipedia.org/w/index.php?title=Levenberg%E2%80%93Marquardt_algorithm&oldid=852940243 (visited on 08/21/2018) (cit. on p. 7).

NVidia (2012). *NVidia Fermi White-paper*. In: *NVIDIAs Next Generation CUDA Compute Architecture*. NVidia (cit. on p. 21).

nvidia.com (2018). *cuFFT CUDA Documentation*. URL: https://docs.nvidia.com/cuda/cufft/index.html.

Power iteration (2018). In: *Wikipedia*. Page Version ID: 851375867. URL: https://en.wikipedia.org/w/index.php?title=Power_iteration&oldid=851375867 (visited on 08/04/2018) (cit. on p. 16).

Pruessmann, K. P. et al. (1999). "SENSE: sensitivity encoding for fast MRI." In: *Magnetic Resonance in Medicine* 42.5, pp. 952–962. ISSN: 0740-3194 (cit. on p. 3).

QtCompanyLtd. (2016a). *QMutex Class*. URL: http://doc.qt.io/qt-4.8/qmutex.html#details.

QtCompanyLtd. (2016b). *Qt Documentation*. URL: http://doc.qt.io/qtcreator/.

QtCompanyLtd. (2016c). *QWaitCondition Class*. URL: http://doc.qt.io/qt-4.8/qwaitcondition.html.

QtCompanyLtd. (2016d). *Using the Meta-Object Compiler (moc)*. URL: http://doc.qt.io/qt-4.8/moc.html.

Rabinowitz, Stanley (n.d.). "How to Find the Square Root of a Complex Number." In: p. 4 (cit. on p. 29).

Rudin, Leonid I., Stanley Osher, and Emad Fatemi (1992). "Nonlinear total variation based noise removal algorithms." In: *Physica D: Nonlinear Phenomena* 60.1, pp. 259–268. ISSN: 0167-2789. DOI: 10.1016/0167-2789(92)90242-F. URL: http://www.sciencedirect.com/science/article/pii/016727899290242F.

Scalar processor (2017). In: *Wikipedia*. Page Version ID: 788096944. URL: https://en.wikipedia.org/w/index.php?title=Scalar_processor&oldid=788096944 (visited on 08/15/2018) (cit. on p. 21).

Steinbach, Peter and Matthias Werner (2017). *FFT Benchmark Suite for Heterogeneous Platforms*. In: *High Performance Computing*. Ed. by Julian M. Kunkel et al. Springer International Publishing, pp. 199–216. ISBN: 978-3-319-58667-0.

Template (C++) (2018). In: *Wikipedia*. Page Version ID: 848963343. URL: https://en.wikipedia.org/w/index.php?title=Template_(C%2B%2B)&oldid=848963343 (visited on 12/03/2018).

Tikhonov regularization (2018). In: *Wikipedia*. Page Version ID: 854154232. URL: https://en.wikipedia.org/w/index.php?title=Tikhonov_regularization&oldid=854154232 (visited on 08/21/2018) (cit. on p. 7).

Uecker, Martin (2009). “Nonlinear Reconstruction Methods for Parallel Magnetic Resonance Imaging.” In: (cit. on pp. 4, 7–9).