MARSHALL PLAN SCHOLARSHIP REPORT

Generalizing Secondary Criterion Improvement for Superiorized Image Reconstruction Algorithms

submitted by:
Christoph Scharf, BSc

Program Director:	FH-Prof. DI Dr. Gerhard Jöchtl
Supervisor SUAS:	FH-Prof. Univ.-Doz. Dr. Stefan Wegenkittl
Supervisor CUNY:	Gabor T. Herman Ph.D.

New York, August 2017

Abstract

The Conjugate Gradient (CG) method is commonly used for the relatively-rapid solution of Least Squares problems. In image reconstruction, the problem can be ill-posed and also contaminated by noise. For this reason an enhanced version of the CG with superiorization for quality improvement, and preconditioning for speed improvement, is proposed.

The mathematical proposition for using conjugated gradient is defined and the PCG algorithm is characterized. The various used filter functions inside the PCG and their effect on the iterative conjugate gradient method are outlined. The evaluation of the PCG, as well as the Superiorized PCG (S-PCG) in comparison to the original CG and the Superiorized CG (S-CG) is done using the image reconstruction framework *SNARK14*. Various filter functions in the PCG are analyzed and the best one is further used for a more detailed investigation of various superiorization parameters and their effect on the PCG. Further, the effect of using more noisy projection data with the CG and the PCG is evaluated.

Contents

List of Figures

1

Iterative Algorithm

Our main interest in using the Conjugate Gradient (CG) as an iterative method is to solve a large linear system of equations. Therefore, the following chapter describes the needed prepositions for the better understanding of the proposed, iterative algorithm. Further superiorization is described, as well as the implementation of it in the software framework *SNARK14*. Different variations of the filter function are proposed and compared against each other in the evaluation in Chapter 2.

1.1 Iterative Conjugate Gradient Algorithm

Due to the large dimensionality of the problems in image reconstruction, direct solving of of the Least Squares Problem is not practical and iterative methods are usually preferred.

The conjugate gradient algorithm successive selects the direction vectors as a conjugate version of the gradients obtained as the method progresses. As an iterative algorithm the methods are not obtained beforehand, but are calculated at each iteration. This allows to approximately solve systems, where n is so large that the direct method would be too computation expensive to directly calculate.

Let us denote the initial starting point for x^* by x_0. By minimizing the quadratic function, the negative of the gradient can be used. The other vectors in the basis will be conjugate to this gradient. Let r_k be the negative gradient, which is in the context

of quadratic functions also called residual, at the k-iteration so that

$$r_k = Qx_k - y. \tag{1.1}$$

At each step k the current negative gradient vector is combined with the previous direction vectors to obtain a new conjugate direction vector along which to move

$$x_{k+1} = x_k + \alpha_k p_k, \tag{1.2}$$

where p_k is the search direction with step α_k. As already discussed before, the efficiency of the conjugate gradient is the fact that each new search direction is conjugated with all previous directions, and the step is the optimal step for minimizing quadratic functions [12, 13, 10].

In general, many iterative algorithms require all previous calculated directions and residual vectors, as well as many matrix vector multiplications and thus can be computationally expensive. However, the iterative conjugate gradient algorithm is based on the conjugate vectors and therefore only r_k, p_k and x_k are needed to construct the next iteration with r_{k+1}, p_{k+1} and x_{k+1}. Although only the last vectors are needed, the new calculated search direction will be Q-orthogonal to all previous search directions. Another main advantage is the especially simple formula needed to calculate the new direction at each iteration step. Also interesting is that most of the progress of reaching the unique solution x^* is made in the first iterations and only slight adjustments are made at the end of all iterations. This is not the case in all iterative algorithms and should be noted that not all iterations are needed to get already a sufficient precise result [12, 15].

Algorithm 1 presents the iterative linear conjugate gradient algorithm with the included steps per iteration. At the end of each iteration at line 7 the current solution is updated, where p_k is the search direction and α_k the step size. The step size is optimal for minimizing the quadratic function. The directions p_k are created according to line 5

$$p_k = -r_k + \beta_k p_{k-1} \tag{1.3}$$

based on the previous direction.

Considering the new point x_{k+1} computationally more efficient by

$$r_{k+1} = r_k + \alpha_k p_k. \tag{1.4}$$

The computation of the new direction also removes parts of the previous direction from the gradient, where β_k is chosen such that

$$p_{k+1}^T Q p_k = (-r_{k+1} + \beta_k p_k)^T Q p_k = 0, \tag{1.5}$$

which shows that the directions are conjugated.

As already shown before, the step α_k is computed as

$$\alpha_k = \frac{-r_k^T p_k}{p_k^T Q p_k}. \tag{1.6}$$

Algorithm 1 Iterative Linear Conjugate Gradient Algorithm

1: $p_0 = 0$

2: **for** $k = 1$ to n **do**

3: $r_k = Qx_k - y$ ▷ New Gradient

4: $\beta_k = \frac{r_k^T Q p_{k-1}}{p_{k-1}^T Q p_{k-1}}$ ▷ Computing beta

5: $p_k = -r_k + \beta_k p_{k-1}$ ▷ New direction

6: $\alpha_k = \frac{-r_k^T p_k}{p_k^T Q p_k}$ ▷ Compute step

7: $x_{k+1} = x_k + \alpha_k p_k$ ▷ New Solution

8: **end for**

1.2 Superiorization

The Superiorization methodology is an automatic and heuristic procedure for taking an iterative algorithm and turning it into a superiorized one. The iterative algorithm

needs to fulfill the properties of being constraint-compatible and resilient to bounded perturbations in order for its superiorized version to produce constraint-compatible solutions [6]. These constraints might be physical properties of the object of interest or any other constraints obtained from any other source. In computerized tomography, the constraints come from the detector readings of the CT scanner [14, 6].

Due to its generic nature, the methodology of superiorization may in principle be applied to any iterative algorithm, which meets the requirement of being resilient to bounded perturbations. It is capable of automatically converting the selected base algorithm into a superiorized version, providing improved results regarding a secondary optimization criterion, which may be freely defined while retaining the original constraint compatibility. This can be any criterion, which evaluates some quality measure of the solution (e.g. measurement of noise). Superiorization incorporates this prior knowledge by steering the solution with bounded perturbations between each iteration of the given algorithm towards a solution, which is superior in the secondary criterion [9, 6, 7, 14, 4].

As an heuristic approach, the superiorization methodology does not guarantee an optimal solution with respect to its secondary criterion. Superiorization will produce a solution, which is both constraint-compatible and also superior with respect to the given secondary criterion. Thus, the superiorized algorithm will produce better solutions, which are as good as the solutions of the original algorithm, but are better in regard to one secondary criterion [6].

Due to the complexity of the method of superiorization and the main focus on the use of the conjugate gradient as the basis method, we will not go in any more details in here. Further information can be found in [6, 7, 14, 4, 9].

The previously discussed properties and requirements of the superiorization are already sufficient to be able to follow the upcoming chapters in terms of applying superiorization to the conjugate gradient and interpreting the results.

1.3 Secondary Optimization Criteria

As discussed in the introduction of superiorization in Chapter 1.2, the secondary optimization criteria can be freely defined. During the conducted experiments and in the use of superiorization within the following experiments, Total Variation (TV) is used as the second order criteria and is defined as

$$TV(x) = \sum_{c \in C} \sqrt{(x_c - x_{p(c)})^2 + (x_c - x_{\xi(c)})^2}, \qquad TV(x) \in \mathbb{R}, \tag{1.7}$$

where C is the set of all indices of pixels (numbered serially and line-by-line) that are not in the rightmost column or the bottom row of the pixel array. For any single pixel x with index $c \in C$, $p(c)$ and $\xi(c)$ are the indices of the pixels to its right and below it [5, 9]. A graphic representation of the kernel function is given in Figure 1.1.

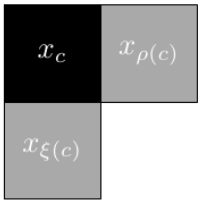


Figure 1.1: Total Variation Kernel Function [9]

1.4 Implementation within SNARK14

SNARK is a software package for reconstruction of 2D images from their 1D projections. *SNARK14* - the current version of it - provides a total framework for reconstruction from projections for both, simulated and real data. It provides the possibility to create projection data for parallel and divergent projection geometry and offers frequently used reconstruction algorithms. The source code is freely available at http://turing.iimas.unam.mx/SNARK14M/ and can easily be extended with user defined functionality as described in [2, 8, 5]. All results presented in Chapter 2 are based

on the software framework *SNARK14*, whereby additional algorithms were added to compare against already implemented methods.

SNARK14 provides the option to turn every iterative algorithm into its superiorized version. To understand the parameter changes in the evaluation Section 2, it requires the knowledge of the perturbation of the length of the direction vector at every superiorization step. The function call for using superiorization in *SNARK14* is

SUPERIORIZE N a b TVAR,

whereby N defines the amount (length) of pertubations added in the superiorization step (1.8), a and b define the length of direction vector defined as

$$\beta = b \times a^l, \tag{1.8}$$

whereby l is a increasing number from 0 to $N-1$ iterations. *TVAR* is defined as the secondary optimization criterion. The exact definition and further information can be found in [8, 5].

1.5 Superiorized Version of the Conjugate Gradient Algorithm (S-CG)

Let us now combine the already discussed method of superiorization in Chapter 1.2 and the iterative conjugate gradient in Chapter 1.1 in the superiorized version of the conjugate gradient algorithm (S-CG). In difference to the already proposed CG, the Algorithm 2 has included the additional line number 4 of the superiorization at each step k. This turns the original CG into the superiorized one.

Algorithm 2 Superiorized Conjugate Gradient Algorithm

1: $p_1 = 0$

2: $x_1 =$ first solution

3: **for** $k = 1$ to n **do**

4: $s_k = superiorization(x_k)$ ▷ New superiorized solution

5: $r_k = Qs_k - y$ ▷ New Gradient

6: $\beta_k = \frac{r_k^T Q p_{k-1}}{p_{k-1}^T Q p_{k-1}}$ ▷ Computing beta

7: $p_k = -r_k + \beta_k p_{k-1}$ ▷ New direction

8: $\alpha_k = \frac{-r_k^T p_k}{p_k^T Q p_k}$ ▷ Compute step

9: $x_{k+1} = s_k + \alpha_k p_k$ ▷ New Solution

10: **end for**

1.6 Superiorized Version of the Precondition Conjugate Gradient Algorithm (S-PCG)

The idea of using a precondition and combining this with the original conjugate gradient algorithm is based on the knowledge, that the reconstructed image includes a lot of noise in the low image frequencies. The idea is to apply a filter part as a precondition at every k step of the conjugate gradient algorithm. Based on the original gradient and the new preconditioned gradient the further steps are calculated. Various filter types used within the Precondition Conjugate Gradient (PCG) are described in the following sections.

1.6.1 Precondition as filter function

As already discussed before the CG algorithm results in some noise in the reconstructed low image frequencies. To remove this unwanted noise in the low frequencies a high-pass filter is applied at every k-step of the iterative CG algorithm. As a result the type and the used parameter of the filter highly effects the reconstruction process, either in

positive or negative way at each step and therefore also the final reconstructed image. This leads to the importance of finding an appropriate filter type and parameter. Section 1.6.3 describes various filter types and their implementation, as well as the evaluation in terms of the residuals in the preconditioned conjugate gradient and the superiorized version as the superiorized preconditioned conjugation gradient.

1.6.2 An additional filtering step

In general, any filtering step, independent of a high- or lowpass filter, can be either done in the space domain or in the frequency domain. Since the design, application and the visualization is much more natural in the frequency domain, we decided to apply the high-pass filter in the frequency domain.

Let us define the frequency filter as Matrix W, as used in (1.9). How this additional calculation step of the new preconditioned conjugate gradient is applied to the iterative CG algorithm, can be seen in Algorithm 3.

$$z = R^T W(Rs - b) \tag{1.9}$$

As already stated before, the main idea is to remove the lower frequencies in the reconstruction process without effecting the higher frequencies. This can be done with a high-pass filter.

1.6.3 Various Filter Functions

In the following Sections 1.6.4, 1.6.5, 1.6.6 and 1.6.7 various high-pass filter functions to remove lower frequencies are described, whereby the frequency filter is denoted as $H(\omega)$ and ω corresponds to the frequency in the frequency domain. In general one has to be aware, that the whole Direct Current (DC) component cannot be eliminated to guarantee a still usable result after the filtering step [15, 3]. This has to be considered when designing the various filters.

In difference to general image processing techniques and algorithms, where mostly two

dimensional filters due to two dimensional images are used, in computed tomographic reconstructions x-rays sums are obtained from the projections along a line in a one dimensional vector. Therefore the proposed filter types in the following sections are all described as one dimensional filters.

1.6.4 Ramp Filter

As the first filter type a Ramp Filter as the simplest high-pass filter is described as

$$H(\omega) = |\omega|. \tag{1.10}$$

As already discussed before, one have to be aware that the complete DC component cannot be eliminated to guarantee a still usable result after the filtering step. Therefore the ramp filter should not totally remove the frequencies at $\omega = 0$. With

$$H(\omega, \mu) = \sqrt{|\omega|^2 + \mu^2} \tag{1.11}$$

depending on a small μ as well, the ramp filter can be shifted towards $H(0) > 0$.

Transforming data from the spatial domain into the frequency domain by using the discrete fourier transformation results in a mirrored signal in the frequency domain. In the implementation of the filter function, (1.11) cannot be used directly, since the data is stored in a one dimensional array of length N. Therefore

$$R(k) = \mu + (\frac{2}{N \cdot \frac{1}{1-\mu}} \cdot (abs(k - \frac{N-1}{2}))) \tag{1.12}$$

can be used, whereby N is the length of array, μ is again the small shift at $H(0) > 0$ and k is the index iterating over the length of the array. This equation is only providing the coefficients of the filter function with the same length as the data-array in the frequency domain. The actual multiplication of the original data in the frequency domain with the calculated filter coefficients has to be done afterwards. Figure 1.2 displays the change of the ramp filter while using various μ parameters.

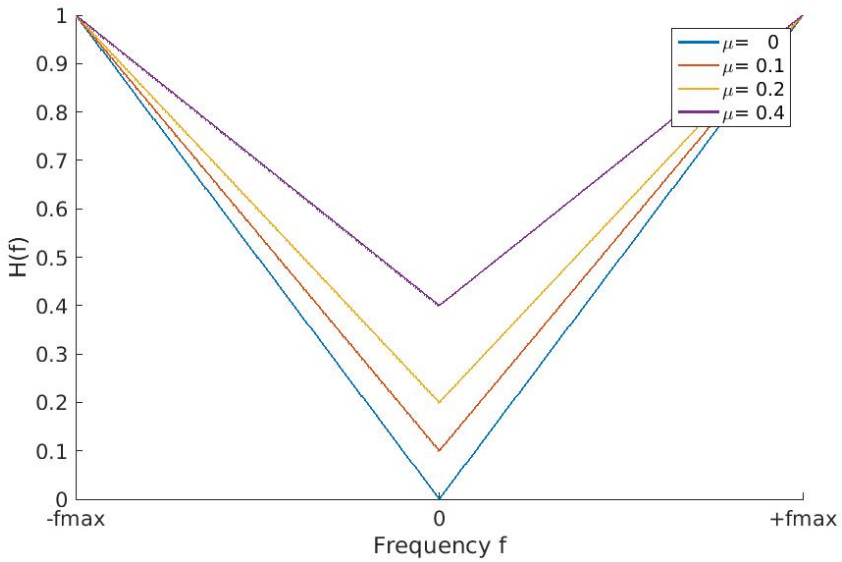


Figure 1.2: Ramp filter with various μ parameters

When using a ramp filter as high-pass filter one has to pay attention to not totally eliminate the DC component at $\omega = 0$. (1.11) is used to guarantee $H(0) > 0$, where H is the filter in the frequency domain and f the frequency. Figure 1.2 displays the change of the ramp filter while using various μ parameters. It can be seen that small μ parameters are already sufficient to not completely remove the DC component[15].

1.6.5 Hann Window

The Hann Window, sometimes noted as Hanning Window is defined as

$$H(\omega) = 0.5 \cdot (1 - \cos(\omega)). \tag{1.13}$$

In difference to the Hamming Window in Section 1.6.6, the end points of the hann window touch zero at the zero-frequency point [11]. Therefore the hann window totally eliminates the DC component and can therefore not be used in the precondition conjugate gradient algorithm.

1.6.6 Hamming Window

As another filter function the Hamming Window is used, which is defined as

$$H(\omega, \alpha) = \alpha - \beta \cdot \cos(\frac{\omega}{N-1}), \tag{1.14}$$

where $\beta = 1 - \alpha$ and N is the length of the one dimensional array. Instead of both constants being equal to 1/2 as in the hann window in Section 1.6.5, the constants are approximations of values $\alpha = 25/46$ and $\beta = 21/46$, which cancel the first sidelobe of the hann window. Therefore the default values are $\alpha = 0.54$, $\beta = 1 - \alpha = 0.46$ [11].

Adjusting the α parameter leads to a different offset at the maximum frequencies of the filter. Different values of α are presented in Figure 1.3. For implementation the filter (1.14) can be used and no specific modifications are needed.

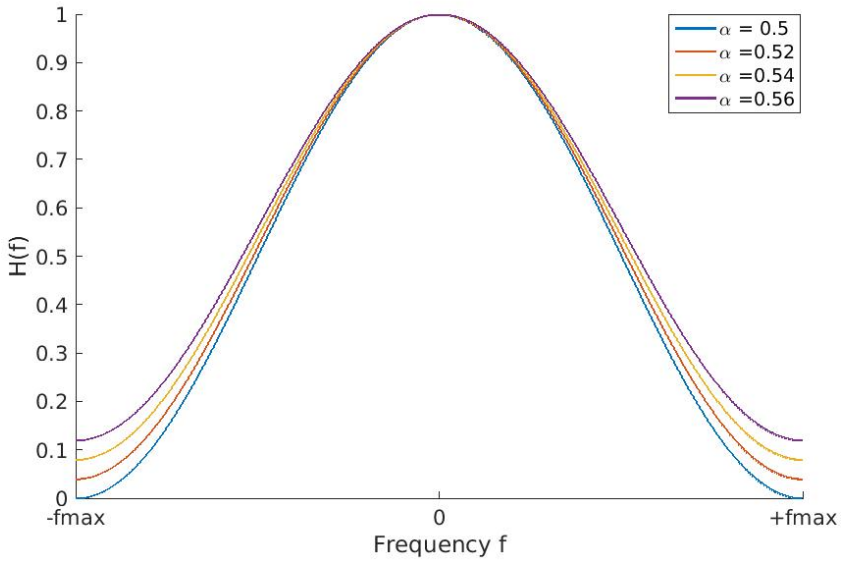


Figure 1.3: Hamming window with various α parameters

When using a hamming window as high-pass filter the zero-frequencies are not eliminated by default. With the change of the α parameter in (1.14) the attenuation at the positive and negative maximum frequency can be adjusted [15, 11].

1.6.7 Combined filter: Hamming with ramp filter function

Another filter function is created by combining the before described ramp function with the hamming window. Therefore

$$H(\omega, \mu, \alpha) = (\omega + \mu) \cdot (\alpha - \beta \cdot \cos(\frac{\omega}{N-1})) \tag{1.15}$$

can be used, where again $\beta = 1 - \alpha$ and N is the length of the one dimensional array. Default values are $\alpha = 0.54$, $\beta = 1 - \alpha = 0.46$ [11]. See Section 1.6.6 for the description of the hamming window and Section 1.6.4 for description of the ramp filter. The Combined Filter function depends on two parameters, the μ, which represents the offset of the ramp filter at the zero frequency, and α, which represents the hamming window offset at the maximum frequencies. The effect of changing the μ parameter in the combined filter function can be seen in Figure 1.5, the effect of varying the α parameter in the combined filter function in Figure 1.6. Figure 1.4 presents the creation and the actual combination of the ramp filter with the hanning window. For implementation purpose, both filter functions have to be multiplied at each frequency ω with each other, as summarized in (1.15).

1.6.8 S-PCG Algorithm Steps

The changes in the algorithm steps in comparison to the original superiorized version of the CG are marked in red in the Algorithm 3, where M is the matrix representing the spatial domain filter and can be replaced by one of the proposed filter functions in Section 1.6.3.

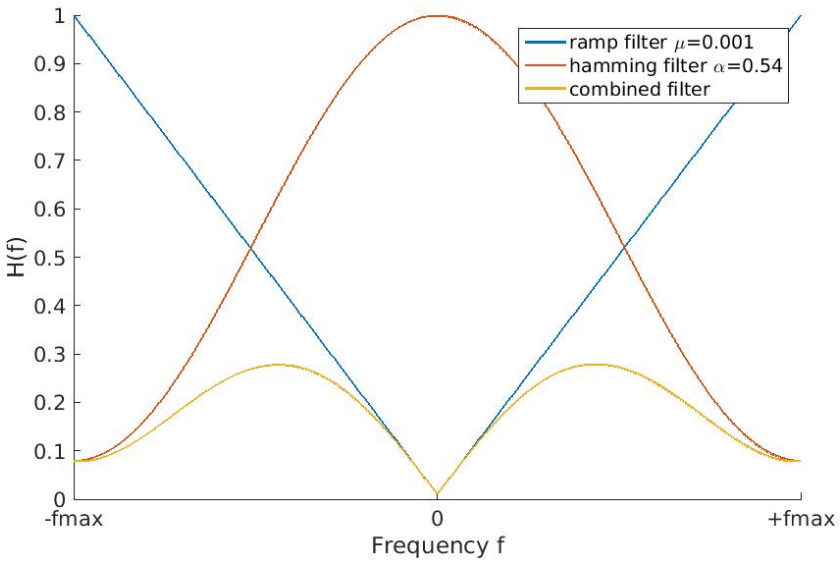


Figure 1.4: Combined filter creation

Figure 1.4 presents the creation of the combination filter based on the ramp filter and the hanning window function. In this example the ramp filter parameter $\mu = 0.01$ and the hanning window parameter $\alpha = 0.54$ are used for demonstration purpose.

Algorithm 3 Superiorized Preconditioned Conjugate Gradient Algorithm

$p_1 = 0$

$x_1 =$ first solution

for $k = 1$ to n **do**

$s_k = superiorization(x_k)$ ▷ New superiorized solution

$r_k = Qs_k - y = R^T Rs_k - R^T b$ ▷ New Gradient

$z_k = R^T WRs_k - R^T Wb$ ▷ New Preconditioned Gradient

$\beta_k = \frac{{z_k}^T Qp_{k-1}}{p_{k-1}^T Qp_{k-1}}$ ▷ Computing beta

$p_k = -{z_k}^T + \beta_k p_{k-1}$ ▷ New direction

$\alpha_k = \frac{-r_k^T p_k}{p_k^T Qp_k}$ ▷ Compute step

$x_{k+1} = s_k + \alpha_k p_k$ ▷ New Solution

end for

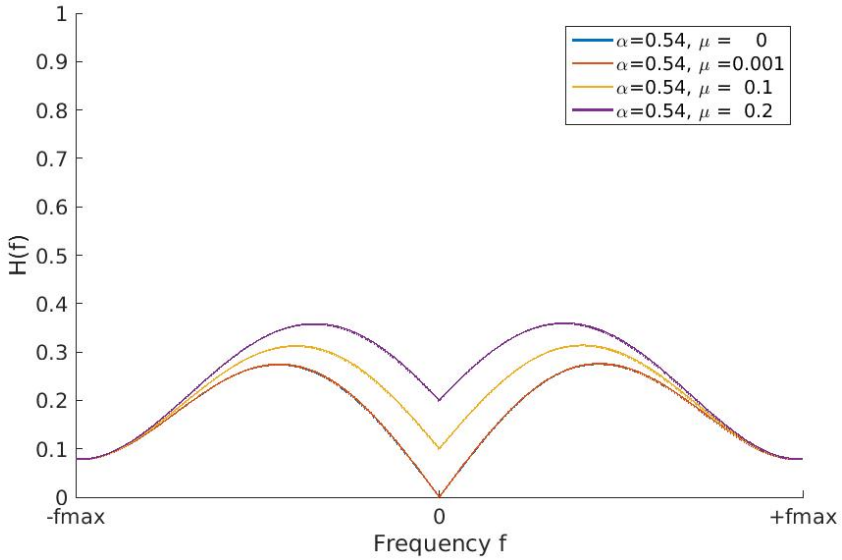


Figure 1.5: Combined filter function with varying μ parameter

Figure 1.5 presents the effects of changing the μ parameter within the combined filter (combination of the ramp with the hamming window filter function). Changing the μ parameter effects the offset at the zero frequency. With $\mu = 0$ the DC component is removed completely, increasing the parameter remains more of the DC components at the zero frequency.

1.6.9 Implementation

To apply the filter in the frequency domain, first the data points have to be transformed into this domain. The *SNARK14* framework provides different functions for transforming data points into the frequency domain and back into spatial domain again [2].

During the implementation of the filter types one has to pay attention to handle the real as well as the complex coefficients after the fourier transformation into the frequency domain to guarantee a usable result after transforming back into the time domain by the inverse fourier transformation [1].

The procedure of applying the high-pass filter or the window functions in the frequency domain is represented and described in more detail in Figure 1.7[1].

The full source code of the conjugated gradient, as well as the preconditioned conjugate gradient algorithm can be found in the Appendix A. Line 95 presents the original conjugates gradient implementation and line 234 presents the preconditioned conjugate

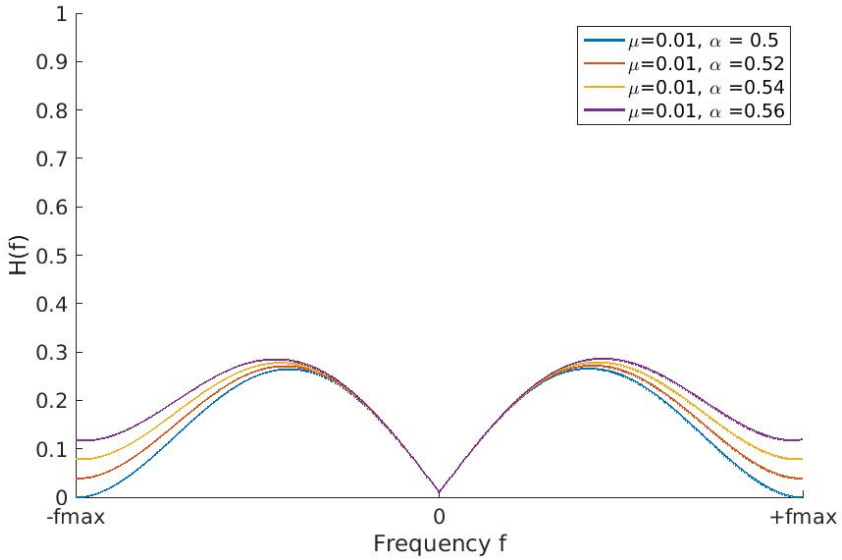


Figure 1.6: Combined filter function with varying α parameter

Figure 1.6 presents the effects of changing the α parameter within the combined filter (combination of the ramp with the hamming window filter function). Changing the α parameter effects the offset at the maximum frequencies of the filter function, but is not effecting the attenuation of the DC component. The default value is $\alpha = 0.54$, approximated on the best removing of side-lobes in the Hann Window function.

gradient. The different implementations of the filter functions are presented as well. The ramp filter can be found in line 150, the hamming window function in line 224 and the combined filter function in line 194.

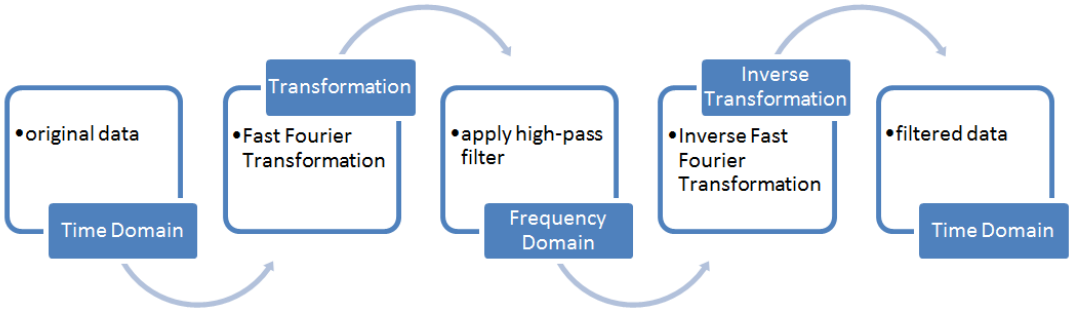


Figure 1.7: Process of filtering in the frequency domain

At first the input data is transformed with the Fast Fourier Transformation (FFT) into the frequency domain, where the filter or window functions, as described in Section 1.6.3, are applied. With the use of the Inverse Fast Fourier Transformation (IFFT) the data is transformed back into the time domain and can further be processed in the algorithm described in more detail in Algorithm 3.

2

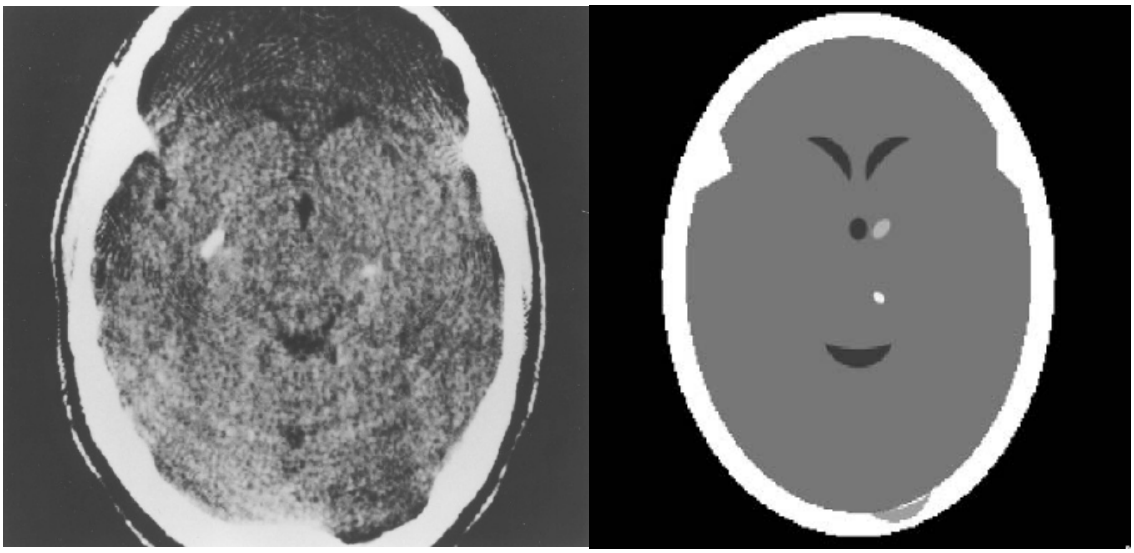
Evaluation

Chapter 2 addresses the evaluation of the proposed methods. All used algorithms are implemented in *SNARK14* as user-defined algorithms. Before the actual evaluation of the proposed algorithms can be done, necessary test data as projection data has to be generated. This is described in the first Section 2.1 and Section 2.2. Section 2.3 describes the conducted experiments and the defined evaluation criteria. Afterwards the evaluation in Section 2.4 is described.

2.1 Phantom as Test Data

SNARK14 is a quite powerful framework, which provides amongst other options the feature to generate specific designed input data to test implemented algorithms and evaluate them based on specific parameters. All this can be done within the *SNARK14* framework. In computed tomographics the input data are phantoms and data projections. The most used phantom is the head phantom. Gabor T. Herman describes in [5] the use of the head phantom in more detail. Thereby the mathematical defined head phantom in Figure 2.1b is based on real data of a reconstructed head of a patient seen in Figure 2.1a. This mathematical exact defined head phantom is mostly used as the basis for algorithm calculations and evaluations in the *SNARK14* framewor,k described in Section 1.4.

Also for the conducted evaluation of the proposed algorithms and various filter described in Section 2, the mathematical defined head phantom is used as basis within



(a) Original x-ray CT reconstruction **(b)** piecewise-homogeneous head phantom

Figure 2.1: Original and mathematically defined head phantom [5]

Image 2.1a shows the central part of a x-ray computer tomography reconstruction of a cross section of the head of a patient. This serves as the basis for the piecewise-homogeneous head phantom. Figure 2.1b presents the mathematically exact defined piecewise-homogeneous head phantom, which is used as the basis for algorithm calculations and methods evaluation within the *SNARK14* framework [5].

SNARK14. Further, the framework *SNARK14* offers many possibilities to fully adjust the mathematically defined head phantom and the belonging x-ray data to individual needs. Therefore, the next Section 2.2 describes the properties of the used head phantom for the conducted evaluation.

2.2 Head Phantom Creation in SNARK14

Section 2.2 describes the properties of the head phantom used in the evaluation part. Therefore, the head phantom, as well as the x-ray sums of a CT device, are simulated and provided by the framework for algorithm evaluation. In the first Section 2.2.1, a head phantom with monochromatic and real world noise for the projection data is generated. The required input commands are provided and described.

2.2.1 Low Noise Projection Data Generation

In Section 2.2.1 the generation of the head phantom with low noise as real world projection data is described. Main part of it is the description of the input file for *SNARK14* to generate the head phantom and the projection data. The used noise level is thereby based on real world noise measurements by real CT devices [5].

Algorithm 4 head phantom generation

```
1: CREATE
2: standard parallel projection data
3: SPECTRUM MONOCHROMATIC
4: 60 100 41 0 52 0 84 0 100 0
5: OBJECTS
6: ...
7: Define various rectangle objects
8: ...
9: LAST 1 0 0.0025
10: PHANTOM AVERAGE 11
11: 243 0.0752
12: RAYSUM AVERAGE 11
13: 1 1 1 1 1 1 1 1 1 1 1
14: GEOMETRY
15: parallel uniform line
16: RAYS user 345 detector spacing 0.0752
17: ANGLES 360 EQUAL SPACING
18: 0.0 179.5
19: MEASUREMENT NOISY
20: QUANTUM 2000000 360 CALIBRATION 1
21: SCATTER 0.012 0.31333333
22: SEED 0
23: BACKGROUND 0.0
```

The *CREATE* command in the first line of Algorithm 4 is used in *SNARK14* to create a phantom with the defined parameters in line 2-23. The second line is a string identifier and is user chosen to identify the generated phantom. The third and fourth lines specify the energy levels used for the x-ray beams. The third line defines that one monochromatic energy level should be used.

The lines 6 to 8 should mark the definition of different objects inside the head phantom, like the objects seen in Figure 2.1b. In difference to the displayed homogeneous head phantom in Figure 2.1b, the head phantom created in Algorithm 4 consists of an

inhomogeneous background area due to command in line 9. The *LAST* command defines that each pixel density d is altered by adding a random sample from a zero-mean normal distribution with standard deviation $0.0025 \cdot d$. Further line 19-21 add noise in different types (quantum and scatter noise) to the the x-ray measurements of the head phantom. A more detailed description of the *SNARK14* commands can be found in [5].

2.3 Conducted Experiments of PCG and S-PCG

As basis for the evaluation of the proposed filter functions in Section 1.6.3 and the proposed reconstruction algorithms, the described head phantoms in Section 2.2 are used. For the conducted experiments the head phantom with real world noise as described in Section 2.2.1 is used.

The Preconditioned Conjugate Gradient (PCG) algorithm with the filter functions as precondition step are compared with the conjugate gradient algorithm within *SNARK14*. The evaluation of the algorithms is done via the residuals of the reconstructed head phantoms and the relative error defined as the secondary order criteria for superiorization in Section 1.3 at every iteration step k. The smaller the relative error and the residuals at each iteration step k, the closer the reconstructed image is to the original one. The closer they are, the less difference is between them, which means that the reconstruction algorithm is obtaining a result as good as the original head phantom. In the conducted experiments the original head phantom is used as the ground truth.

Thereby one has to differentiate between the first and the second order criteria as described in Chapter 1.2. Without superiorization only the residuals, as the first order criteria, are taken into consideration. Applying superiorization should improve only the second order criteria, as the relative error, without influencing the first order criteria. Thereby superiorization does not find the exact minimum solution of the second order criteria. It minimizes the second order criteria without exacerbating the first order criteria. Superiorization does not mess up the first order criteria and is therefore not altering the convergences of the algorithm. Due to that, an algorithm, which converges without superiorization, still converges using superiorization.

In the evaluation in Section 2.4 various filter functions in the PCG are compared with each other by considering the residuals as well as the relative error.

While superiorization delivers better second order criteria, it is also influencing the original algorithm. To gain insights of the PCG without being effected by the superiorization step, the first experiment is conducted without superiorization.

In the second experiment in Section 2.5, the PCG is taken into its superiorized version. Maintaining an overview and not resulting in too many plots in one figure, only the best filter function of the original PCG of the first experiment is taken into its superiorized version. In this second experiment the superiorization parameters are varied and their effect on the relative error and residuals is considered. Another goal at this step is finding the best parameter for using superiorization.

In general two main criteria are defined to classify the quality of an algorithm. In the first place, the relative error should be getting as low as possible. A small relative error relates to a small difference between the original head phantom as the ground truth and the reconstructed head phantom at each iteration step k. Although the algorithm might only reach the lowest relative error for only a few iterations and raises again afterwards, the algorithm is still considered as a good one. In the second place, the number of needed iterations to reach the minimum relative error should be as less as possible. Fewer iterations means receiving the best results in less time, since every iteration also needs computation time.

Another interesting question in working with iterative algorithm is at which point the algorithm should be stopped to achieve the best results. This criteria for stopping is called *stopping criteria*. In computed tomographic reconstructions and while using *SNARK14* the stopping criteria in the conducted experiments is reached, when the residuals of the reconstruction pass the residuals of the original head phantom. The residuals of the original head phantom are marked as a horizontal red line in all conducted experiments. Different than one would expect, the residuals of the original head phantom are not zero. This is based on the existing difference between the mathematical precisely defined head phantom and the digital created head phantom due to quantization error and non endless precise values when working in the digital world.

2.4 Evaluation 1: PCG with various Filter function

In the evaluation the various filter functions described in Section 1.6.3 and their impact on the PCG algorithm steps, described in Section 1.6.8, are evaluated and compared to the original CG algorithm. To investigate the effect of the various filter functions while using the preconditioned conditional gradient (PCG), the basis form without superiorization is used. Therefore, the algorithm is not influenced by the superiorization step at every iteration k.

Figure 2.4 displays the residuals, Figure 2.2 the relative error of the PCG with using various filter functions as described in Section 1.6.3. At every k-step the respective filter type is applied at the CG, which results in the PCG presented in Algorithm 3 excluding the superiorization step in line 4.

For better insights in the first iterations of the relative error, the first $k = 36$ iterations are plotted in Figure 2.3. The used filter type and parameters for the filter functions can be found in the legend of the figures. The ramp and hamming window are characterized by one parameter, e.g δ or α, the combined filter function is characterized by the combination of both of them.

In general, it can be argued that the type of used filter function in the PCG highly impacts the residuals and the relative error during the reconstruction process. It can be seen that the ramp filter with $\delta = 0.0001$ achieves quite low relative error in less iterations, but the relative error is raising quickly after reaching the minimum. The hamming window with $\alpha = 0.54$ achieves low relative error, but needs more iterations. Despite the expectations of achieving better relative error while combining both of these parameters in the combined window function with $\delta = 0.001, \alpha = 0.54$, it does not deliver better results. In contrast, increasing the α parameter to higher values improves the relative error. Therefore, using the combined window function with $\delta = 0.0001, \alpha = 0.80$ accomplishes the lowest residuals in the fewest needed iterations. One could also notice that changing the δ parameter below 0.001 does not effect the outcome of the relative error. While setting the parameter to a higher value, like $\delta = 0.1$ results in significant higher relative error values.

Analyzing the relative error in terms of the before defined evaluation criteria at the end

of Section 2.3 (resulting in low relative error in few iterations), one can see that the PCG does not achieve a relative error as low as the original CG algorithm. Apart from this, the difference in the relative error is only 0.0016, but can already be achieved after two iterations. This is an improvement of 13 less iteration steps. For most applications and reconstruction algorithms reconstruction time is an important factor and therefore the proposed PCG algorithm can be used as an alternative to gain a significant time improvement.

Figure 2.4 presents the plot of the residuals of the various filter functions used in the PCG. As already described at the end of Section 2.3 the stopping criteria is defined as passing the residuals of the head phantom and is marked red. With this stopping criteria, the PCG using the ramp filter marked as blue line, stops already after the second iteration, whereby the CG algorithm, marked in green, is reaching the stopping criteria as late as at iteration 18.

In general, it can also be seen that the residuals of all PCG functions raise quickly after reaching the minimum point. To counteract this effect, superiorization can be applied at each iteration step to improve the stability of the second order criteria defined as the relative error. The evaluation in Section 2.5 addresses the use of superiorization with various parameters while using the PCG.

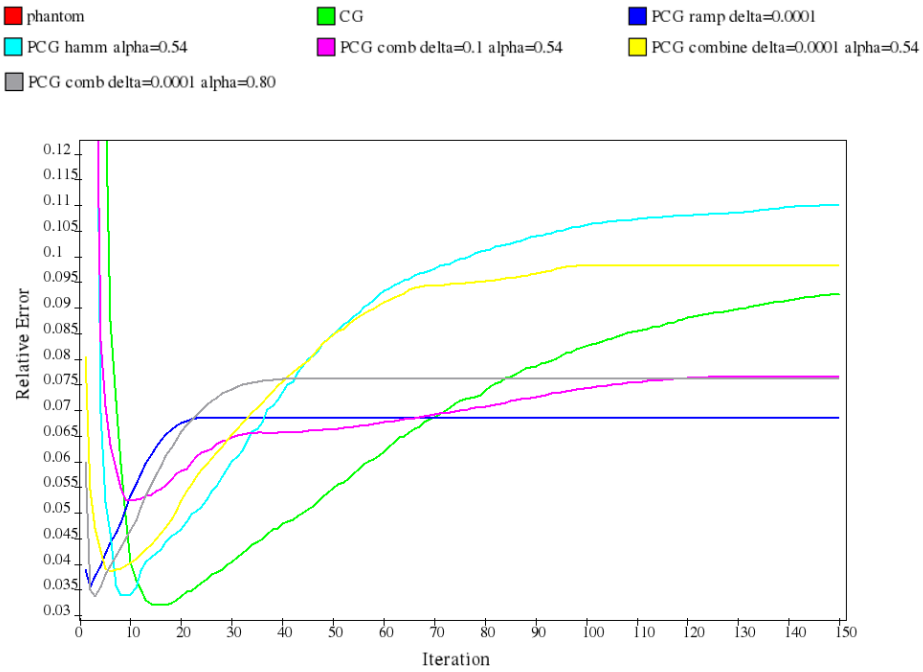


Figure 2.2: Relative error of PCG with different filter functions up to $k = 150$

The relative error of the PCG using various filter functions is plotted in Figure 2.2. The used parameter for the plots can be found in the legend. For the ramp and the hamming window one parameter, δ or α is needed, the combined filter function, as a combination of both of them, is defined by combining both parameters and both filter functions as seen in Figure 1.4. The filter functions and examples are described in Section 1.6.3. For better insights in the first iterations, the first $k = 36$ iterations of the relative error are plotted in Figure 2.3.
In general it can be argued, that the type of filter function used in the PCG highly impacts the residuals and the relative error during the reconstruction process. Retaining too much of the DC component during the reconstruction process, as with using the combined filter with $\delta = 0.1, \alpha = 0.54$, is resulting in the same high relative error as using a combined filter function with a too low α parameter. During the evaluation process it could be noticed that varying the δ parameter at values lower than $\delta < 0.001$ does not effect the relative error anymore.
Analyzing the different filter types and used parameters in terms of the defined evaluation criteria at the end of Section 2.3, it can be seen, that the PCG does not achieve relative errors as low as the original CG. However it can also be seen that the difference is small. A big advantage is the significant reduced number of needed iterations to reach the minimum point when using the PCG instead of using the CG. For most applications and reconstruction algorithms the reconstruction time is a critical factor and therefore the proposed PCG algorithm provides a significant improvement in the needed reconstruction time.

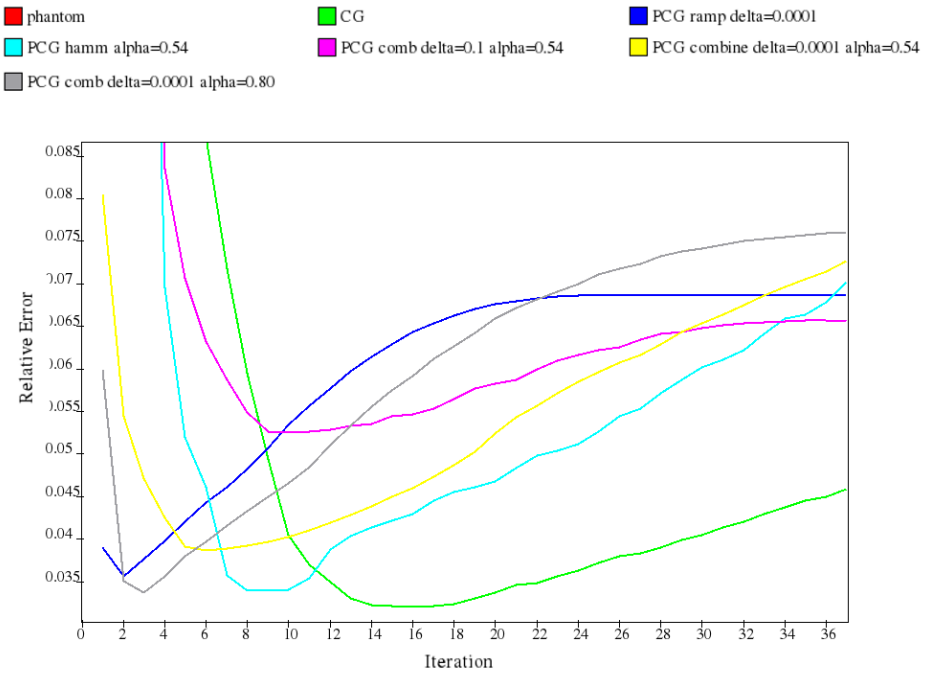


Figure 2.3: Relative Error of PCG with various filter functions up to $k = 36$

Figure 2.3 provides a more detailed plot of the relative error for better insights in the first iterations of the various filter functions. All iterations and a more precise description of the figure can be found in Figure 2.2.

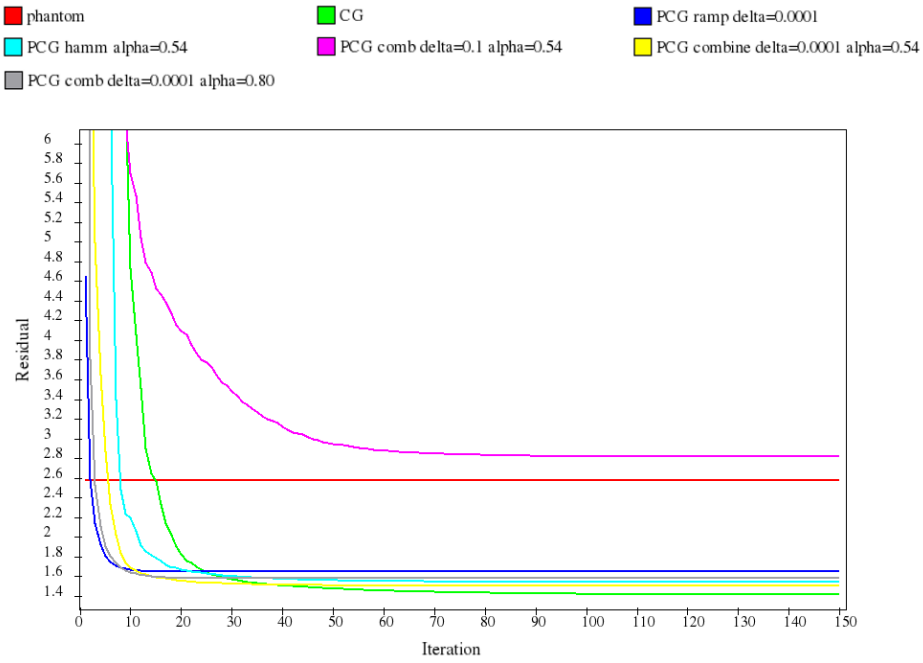
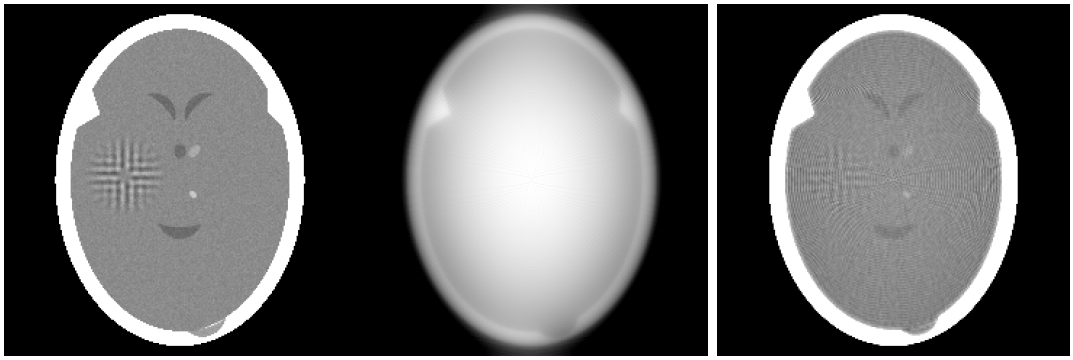


Figure 2.4: Residuals of PCG with various filter functions up to $k = 150$

Figure 2.4 presents the residuals of the PCG while using different filter functions and the original CG. As already seen in the relative error in Figure 2.2, the used filter function and used parameters highly effects the process of reconstruction - either in positive or negative way. The used filter parameter of the plots can be found in the legend of the figure.
It can be seen in the plot of the residuals, as well as in the plot of the relative error, that the residuals of the PCG do not decrease as low as the residuals of the original CG. Although there is a difference existing, the difference of the relative error is smaller than 0.0016. In contrast, it can be noticed that 13 fewer iterations are needed to achieve the almost same residuals. For most applications and reconstruction algorithms reconstruction time is a critical factor. Therefore, the proposed PCG algorithm can be used as an alternative, which provides a significant improvement in the needed reconstruction time.

For better visualization of the residuals and relative error and especially their meaning and effects on the head phantom, Figure 2.5 presents a comparison of the original head phantom (Figure 2.5a), the reconstructed head phantom using CG (Figure 2.5b) and the reconstructed head phantom using the PCG and the combined filter function with $\delta = 0.0001, \alpha = 0.54$ (Figure 2.5c). Both reconstructions are captured after the first iteration $k = 1$. The combined filter and the parameters are chosen due to the smallest residuals in the previous evaluation in Figure 2.4. It can be seen that the head phantom

of the CG has a significant lower sharpness and quality in comparison to the PCG at the same iteration step $k = 1$. This observation corresponds to the discussed residuals in Figure 2.4 and the discussed relative error in Figure 2.2. Within these figures it can be seen that the residuals and the relative error are significantly smaller using the PCG in the first iteration as using the CG algorithm.



(a) Original head phantom **(b)** Reconstruction of CG **(c)** Reconstruction of PCG

Figure 2.5: Comparison of reconstructed head phantoms of CG and PCG at $k = 1$

Figure 2.5 presents the original and the reconstructed head phantom using the CG and the PCG algorithm. Figure 2.5a displays the original head phantom, Figure 2.5b presents the reconstructed head phantom of CG and Figure 2.5c presents the reconstructed head phantom using the combined filter function with parameters $\delta = 0.001$ and $\alpha = 0.54$. Both reconstructed head phantoms are captured after the first iteration $k = 1$. The filter function and parameter for the PCG are chosen due to the smallest relative error in Figure 2.2. It can definitely be seen that the CG reconstructed head phantom has a lower sharpness and quality compared to the PCG at the same iteration step $k = 1$. This observation corresponds to the relative error in Figure 2.2, where the relative error of the PCG is significantly smaller than the CG after the first iteration $k = 1$.

2.5 Evaluation 2: Varying Superiorization Parameter

In this experiment of Section 2.5 a more detailed look is taken on varying the superiorization parameter and evaluating the effect on the relative error during the reconstruction process. Based on the results of the first evaluation in Section 2.4 only the combined filter function with one of the lowest relative errors (parameter $\delta = 0.0001, \alpha = 0.54$) is taken as the best representative filter for the PCG. Although

these parameters do not result in the lowest relative error, observations using superiorization have shown that the combined filter function in the PCG with parameter $\delta = 0.0001, \alpha = 0.54$ delivers better results than the best relative error filter function of Figure 2.3 with parameter of $\delta = 0.0001, \alpha = 0.80$). Therefore $\delta = 0.0001, \alpha = 0.54$ are used as representative parameters for further experiments when using the PCG.

Generally speaking, the superiorization of an algorithm should not change the first order criteria, but should improve the algorithm in terms of the defined second order criteria. In this case the first order criteria is defined as the residuals, the second order criteria as the relative error. Thereby the relative error describes the difference between the original head phantom and the reconstructed head phantom and should be used as criteria for algorithm comparison. A more detailed description of superiorization can be found in Chapter 1.2.

As already discussed in the beginning of Section 2.3, two main evaluation criteria are defined. At first, the relative error should be getting as small as possible. Secondly, the minimum relative error should be achieved within few iterations. As seen in Figure 2.2, the relative error using the CG and the PCG without superiorization tends to raise quickly after reaching the minimum point. Besides of reaching lower second order criteria, the aim of superiorization and the chosen superiorization parameters is to steer the relative error to stay at the level of the local minimum for a longer time of iterations. This could be defined as a minor, third evaluation criteria.

As already discussed in Section 1.4 any algorithm inside *SNARK14* can be taken into its superiorized version with the input command of

SUPERIORIZE N a b TVAR,

whereby the three parameters define the steering of superiorization. A more detailed description of these parameter can be found in the beginning of Section 1.4.

In this evaluation setup of using superiorization, the number of iterations N inside the superiorization step is set to constant $N = 15$ for all conducted experiments. Further, the parameters a, b represent a two dimensional minimization function to find the best relative error for the S-PCG. To solve the two dimensional minimization problem in

a suitable amount of time, the fist parameter a is varied, b is set fixed and the a parameter delivering the best relative error in terms of the defined evaluation criteria over k iterations is found. Figure 2.6 presents the plot of the relative error with varying the a parameter. Afterwards, the b parameter is varied, whereby the a parameter is set to the best value identified beforehand. During the conduction of the experiment it could be noticed that changing b is effecting the residuals only in a very small range. Therefore, the b parameter is set to the fixed value of $b = 1.5$ and only the best a parameter has to be investigated.

Figure 2.6 presents the relative error of the S-CG and the S-PCG with varying superiorization parameter a. The parameter for the CG algorithm was fixed to $a = 0.990, b = 1.5$ due to the best results of Marcelo V. W. Zibetti [10]. The parameter for the PCG are chosen as best representative to demonstrate the behavior of varying the a parameter. It can be seen that using a too small $a = 0.970$, the superiorization cannot steer the algorithm to stay at the reached minimum up to iteration $k = 150$. Although using an $a = 0.999$ as close to 1, the S-PCG does not result in the lowest relative error, which can be reached by using $a = 0.990$. Further, it can be seen that using superiorization, the algorithm is keeping the lowest relative error up to $k = 150$ iterations.

The opposite results can be noticed when having a closer look at the residuals of the S-CG and the S-PCG with varying a parameter in Figure 2.7. It can be seen that once the superiorization is starting to steer the algorithm stronger between iteration 50-60, the low a values with the high relative error results in lower residuals. While considering the stable relative error plot with $a = 0.990$ it can be noticed that the corresponding residuals are also stable. Total stability, but not the lowest relative error, as well as not the lowest residuals, can only be reached while using an a parameter close to 1 ($\alpha = 0.999$).

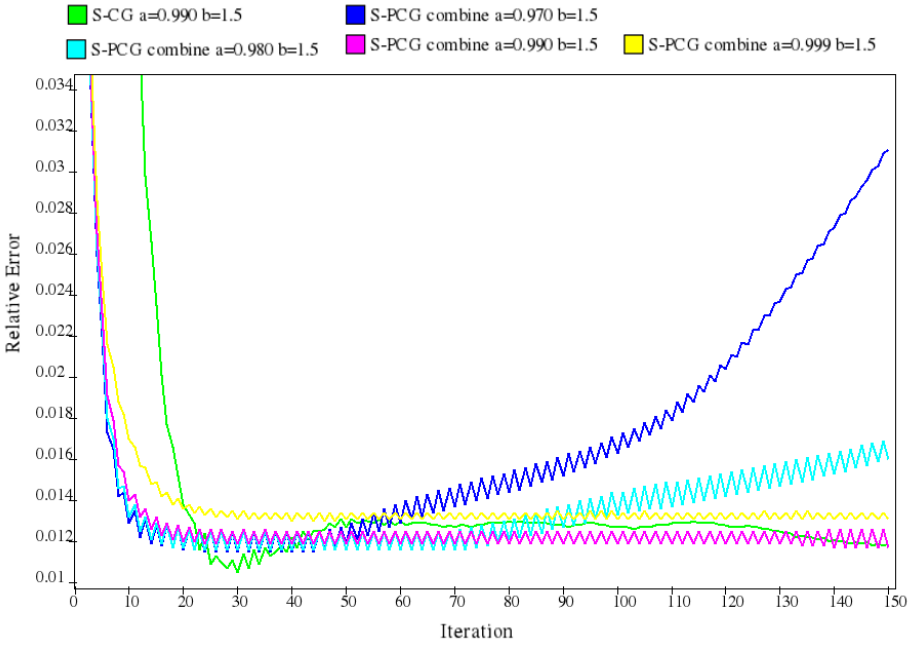


Figure 2.6: Relative error of S-PCG with varying parameter a

Figure 2.6 presents the relative error of the S-CG and the S-PCG with varying superiorization parameter a. It can be seen that using a too small superiorization parameter a (e.g. $a = 0.070$), superiorization cannot steer the algorithm to stay at the reached minimum for $k = 150$ iterations. The best results can be achieved while using $a = 0.990$.

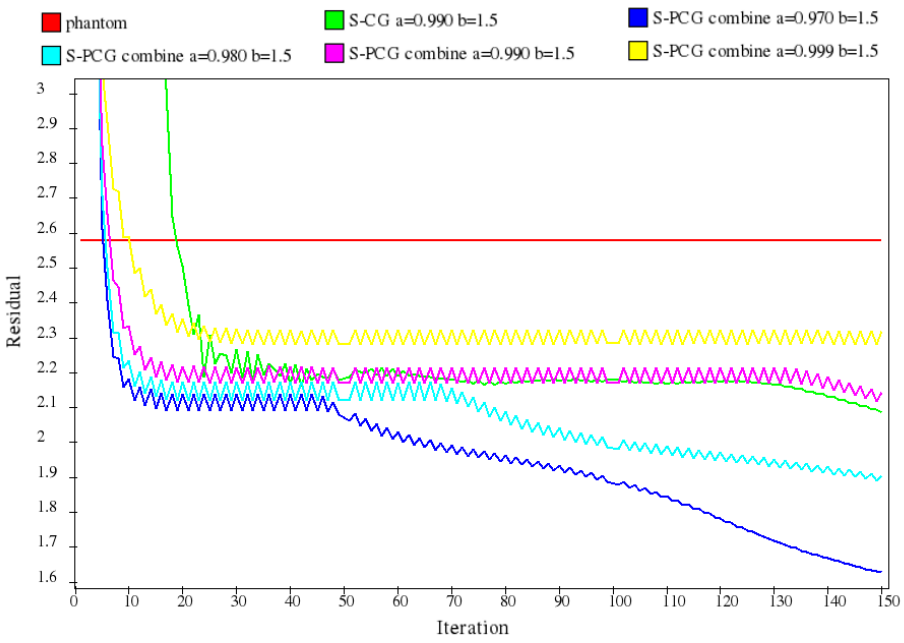


Figure 2.7: Residuals of S-PCG with varying parameter a

Figure 2.7 presents the plot of the residuals using the S-CG and the S-PCG with varying a parameter. In difference to the relative error in Figure 2.6, the opposite results can be noticed when having a closer look at the residuals of the S-PCG. It can be seen that once the superiorization starts to steer the algorithm stronger between iteration 50-60, the low a values with the high relative error results in lower residuals. While considering the stable relative error plot with $a = 0.990$, it can be noticed that this one is also quite stable in terms of the residuals. Total stability, but not resulting in the lowest relative error, as well as not resulting in the lowest residuals, is only achieved when using an α parameter close to 1 ($\alpha = 0.999$).

3

Conclusion

In the first chapter the iterative conjugate gradient algorithm, the superiorization including the primary and secondary criteria and the implementation of them is described. Further the combination of superiorization and the iterative algorithm is characterized. The proposed precondition conjugate gradient algorithm is described as well as different filter functions. The implementation is done using the *SNARK14* framework, which already provides a powerful toolkit for computed tomography reconstruction algorithms, among other the possibility to turn an iterative algorithm into its superiorized version. The PCG implementation is roughly described and the source code within the *SNARK14* user defined functions is provided.

Based on the previous described and proposed algorithms and methods, the evaluation sections provide a discussion of the proposed preconditioned conjugate gradient and compares it with the original conjugated gradient algorithm.

Within the experiments the original conjugate gradient algorithm is taken as reference algorithm for comparison with the proposed preconditioned conjugate gradient. Thereby various filter types and filter parameter in the preconditioning step are taken into consideration during the evaluation of the algorithm. As measurement and objective metric for algorithm comparison, the residuals and the relative error per iteration are used and described in more detail.

As explained before, the first experiment considers both algorithms in their original version. In the second experiment both algorithm are superiorized by the use of the *SNARK14* framework. Therefore the Superiorized Conjugate Gradient (S-CG), as

well as the Superiorized Preconditioned Conjugate Gradient (S-PCG) are compared in terms of the residuals and the relative error per iteration.

Within the evaluation of the superiorized algorithm comparison, the filter type and parameters of the preconditioned conjugate gradient are set as constant and only the superiorization parameters are differed. The effects on the residuals and relative error per iteration are discussed and described.

During the conduction of all experiments it could be seen that the type of used filter function and filter parameter in the preconditioning step of the preconditioned conjugate gradient highly influences the results during the reconstruction process.

Therefore one has to be aware, that the type of the applied filter function highly effects the result of the reconstruction process in either a positive, or also negative way and one has to choose the filter type wisely.

Bibliography

[1] Bracewell, Ronald Newbold and Bracewell, Ronald N: *The Fourier Transform and its Applications*. McGraw-Hill Kogakusha, Ltd., Tokyo, second edition, 1978.

[2] Davidi, R, Herman, Gabor T, Langthaler, Oliver, Sardana, SAUMYA, and Ye, Z: *Snark14: a programming system for the reconstruction of 2d images from 1d projection*. 2017.

[3] Gonzalez, Rafael C., Woods, Richard E., and Eddins, Steven L.: *Digital Image Processing Using MATLAB*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003, ISBN 0130085197.

[4] Havas, Clemens: *Superiorization of the ml-em algorithm with various secondary criteria*. Master thesis, Salzburg University of Applied Sciences, 2016.

[5] Herman, Gabor T: *Fundamentals of computerized tomography: image reconstruction from projections*. Springer Science & Business Media, 2009.

[6] Herman, Gabor T, Garduño, Edgar, Davidi, Ran, and Censor, Yair: *Superiorization: An optimization heuristic for medical physics*. Medical Physics, 39(9):5532–5546, 2012.

[7] Janakiev, Nikolai: *Superiorized algorithms for medical image processing: Comparison of shearlet-based secondary optimization criteria*. Master thesis, Salzburg University of Applied Sciences, 2017.

[8] Klukowska, Joanna, Davidi, Ran, and Herman, Gabor T: *Snark09–a software package for reconstruction of 2d images from 1d projections*. Computer methods and programs in biomedicine, 110(3):424–440, 2013.

[9] Langthaler, Oliver: *Incorporation of the superiorization methodology into biomedical imaging software.* Master thesis, Salzburg University of Applied Sciences, 2014.

[10] Marcelo V. W. Zibetti, Chuan Lin, Gabor T. Herman: *Total variation superiorized conjugate gradient method for image reconstruction.* to appear in Wiley Interdisciplinary Reviews: Computational Statistics, 2017.

[11] MATLAB: *version 8.5.0.197613 (R2015a).* The MathWorks Inc., Natick, Massachusetts, 2015.

[12] Nazareth, JL: *Conjugate gradient method.* Wiley Interdisciplinary Reviews: Computational Statistics, 1(3):348–353, 2009.

[13] Nocedal, Jorge and Wright, Stephen J: *Conjugate gradient methods.* Numerical optimization, pages 101–134, 2006.

[14] Prommegger, Bernhard: *Comparison of iterative algorithms with and without superiorization for image reconstruction from projections.* Master thesis, University of Salzburg, University of Applied Sciences Salzburg, 2014.

[15] Zibetti, Marcelo V. W.: *Superiorization applied to conjugate gradient in image reconstruction.* Technical report, City University of New York, Research Community, Internal report, 2017.

Appendix

A

User Defined Functions of SNARK14 for Preconditioned Conjugate Gradient

```
1  /*
2   * * * * * * * * * * * * * * * * * * * *
3   *
4   *             S N A R K   0 9
5   *
6   *     A PICTURE RECONSTRUCTION PROGRAM
7   *
8   * * * * * * * * * * * * * * * * * * * * *
9
10  alp2.cpp,v 1.3 2009/06/01 03:33:25 jklukowska Exp
11
12  Implemented a conjugate gradient planned to be utilized
       with superiorization
13  V1.0 2016/08/25 by Marcelo V. W. Zibetti
14
15  Updated by Christoph Scharf, Februar 2017
16  Preconditional Conjugate Gradient with different filter
       functions
17     selectable by the input file
18
19 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
20 C
21 C THIS IS A CONJUGATE GRADIENT ALGORITHM TO BE UTILIZED
      WITH SUPERIORIZATION
22 C
23 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
24 //testing making changes. Ran.
25 */
26
27 #include <cstdio>
28 #include <cmath>
29 #include <iostream>
30 #include <iomanip>
31 #include <fstream>
32 #include <cstring>
33 #include <cstdlib>
34 #include <limits>
35
36 #include "blkdta.h"
37 #include "uiod.h"
38
```

```
39 #include "alp2.h"
40
41 #include "geom.h"
42 #include "misl.h"
43 #include "wray.h"
44 #include "raysel.h"
45 #include "consts.h"
46 #include "infile.h"
47 #include "pick.h"
48 #include "anglst.h"
49 #include "second.h"
50 #include "term.h"
51 #include "snfft.h"
52 #include "rtfort.h"
53
54
55 //
    ==========================================================================

56 // This function computes the inner product res=p'*x
57 REAL alp2_class::inner(REAL* p,REAL* x)
58 {
59   REAL sum;
60
61   sum = 0.0;
62   for(uint i=0;i<GeoPar.area;i++)
63     sum += p[i]*x[i];
64
65   return sum;
66 }
67
68
69 //
    ==========================================================================

70 // This function computes the new direction  p = -g + b*p
71 void alp2_class::comp_p(REAL* p,REAL* z,REAL beta_k)
72 {
73   for(uint i=0;i<GeoPar.area;i++)
74   {
75       p[i] = -z[i] + p[i]*beta_k;
76   }
77 }
78
79
80 //
    ==========================================================================

81 // This function computes the updated image x = x + a*p
82 void alp2_class::comp_x(REAL* x,REAL* p,REAL alpha_k)
83 {
84   for(uint i=0;i<GeoPar.area;i++)
85   {
86       x[i] = x[i] + p[i]*alpha_k;
87   }
88 }
89
90
```

```
91 //
     ==========================================================================
92 // This function computes gradient for the conjugate
     gradient algorithm: r=R'*(R*x - y)
93 //grad_x(g,list,weight,recon);
94
95 void alp2_class::grad_x(REAL *unfiltered, REAL *z, INTEGER*
     list, REAL* weight, REAL *x)
96 {
97   REAL val;
98   REAL tsum;
99   INTEGER numb;
100  REAL snorm;
101
102  memset(z,0,GeoPar.area*sizeof(REAL));
103  memset(unfiltered,0,GeoPar.area*sizeof(REAL));
104
105  // R = weightin
106  // x = x, values of ray
107  // y = Anglst.prdta
108  // R' difference to R
109
110  if(!GeoPar.strip){
111    // LINE MODE
112    for(INTEGER np = 0; np < GeoPar.prjnum; np++){
113      for(INTEGER nr = 0; nr < GeoPar.nrays; nr++){
114        wray(np, nr, list, weight, &numb, &snorm);
115        if(numb > 0){
116          val = 0.0;
117          tsum = 0.0;
118          // val = R*x -y
119        //Iterate over all pixels of the trace LINE ray
120        //weight the lengths of the ray segments
121          for(INTEGER nb = 0; nb < numb; nb++){
122            tsum += x[list[nb]] * weight[nb];
123          }
124          val = tsum - Anglst.prdta(np,nr);
125          // r = R'*(R*x -y)
126          for(INTEGER nb = 0; nb < numb; nb++){
127            unfiltered[list[nb]] += weight[nb]*val;
128            z[list[nb]] = unfiltered[list[nb]];
129          }
130        }
131      }
132      //finished one image
133
134      //apply filtering in the spatial domain
135     // filter(unfiltered, filtered, GeoPar.area);
136    } //finished all projections
137  }
138 }
139
140 //Triangular Filter Function
141 void alp2_class::getTriangularFilter(int n, double addToL,
     double* filterValues)
142 {
143     double L = n + addToL;
```

```
144
145     for(int i = 0; i < n; i++)
146     {
147         filterValues[i] =  1 - fabs( (i - (n-1)/2) / (L/2)
              );
148     }
149 }
150 //Ramp Filter Function
151 void alp2_class::getRampFilter(int n, double mu, double*
      filterValues)
152 {
153     double rampFilterValue = 0;
154
155     for(int i = 0; i < n; i++)
156     {
157         rampFilterValue = 1 -  ( (2.0 / (n* ( 1.0/(1.0-mu)
              )) ) * ( fabs((double)i-(((double)n-1)/2.0))) ) ;
158         filterValues[i] = rampFilterValue;
159     }
160 }
161 //combined fiilter function
162 void alp2_class::getCombFilter(int n1, double mu, double
      alpha, double* filterValues)
163 {
164
165     double beta = 1.0 - alpha;
166
167     double rampFilterValue = 0;
168     double hannfilterValue = 0;
169     double adjustRampOffset = 0;
170
171     double n = n1*1.0;
172
173     for(int i = 0; i < n; i++)
174     {
175         rampFilterValue = 1 -  ( (2.0 / (n* ( 1/(1-mu) )) )
              * ( fabs((double)i-((n-1)/2.0))) ) ;
176
177         hannfilterValue = 1 -  ( alpha - beta * cos( (2.0*
              Consts.pi*(double)i) / n ));
178
179         if(i == 0)
180         {
181             adjustRampOffset = 1.0 - hannfilterValue;
182         }
183
184         hannfilterValue = hannfilterValue +
              adjustRampOffset;
185
186
187         filterValues[i] = rampFilterValue * hannfilterValue
              ;
188     }
189 }
190
191 //Hamming window function
192 void alp2_class::getHammFilter(int n, double alpha, double*
      filterValues)
```

```
193 {
194     double beta = 1.0 - alpha;
195 
196     for(int i = 0; i < n; i++)
197     {
198         filterValues[i] =  alpha - beta * cos( (2.0*Consts.
              pi*(double)i) / (double)n );
199     }
200 }
201 
202 void alp2_class::grad_x_filter(REAL *filtered, REAL *
      unfiltered, INTEGER* list, REAL* weight, REAL *x)
203 {
204     fprintf(output,"Starting with grad_x\n");
205 
206     double val;
207     double tsum;
208     int numb;
209     double snorm;
210     double yValue;
211 
212     int nelem = GeoPar.nrays*2;
213 
214     int *n = new int [3];
215     memset(n,0,3*sizeof(int));
216 
217     n[0] = GeoPar.nrays;
218     n[1] = 1;
219     n[2] = 1;
220 
221     raySum = new REAL[nelem*sizeof(REAL)];
222     raySumFiltered = new REAL[nelem*sizeof(REAL)];
223 
224     double *filterValues = new double [GeoPar.nrays];
225 
226 
227     memset(unfiltered,0,GeoPar.area*sizeof(REAL));
228     memset(filtered,0,GeoPar.area*sizeof(REAL));
229     memset(filterValues,0,GeoPar.nrays*sizeof(REAL));
230 
231 
232     if(!GeoPar.strip)
233     {
234         //Get the filter values here, since they stay the
              same for the whole iterations
235         if(windowFunction == ramp)
236         {
237             getRampFilter(n[0], rampDelta, filterValues);
238         }
239         else if(windowFunction == hamm)
240         {
241             getHammFilter(n[0], hamm_alpha, filterValues);
242         }
243         else if(windowFunction == triangular)
244         {
245             getTriangularFilter(n[0], triangular_L,
                  filterValues);
246         }
```

```
247         else if(windowFunction == combination)
248         {
249             getCombFilter(n[0], comb_mu, comb_alpha,
                    filterValues);
250         }
251
252         //console1D(filterValues, n[0], "filterValues");
253         //consoleComplex(raySumFiltered, nelem, np, "
                complex");
254
255         // LINE MODE
256         for(INTEGER np = 0; np < GeoPar.prjnum; np++)
257         {
258             memset(raySum,0,nelem*sizeof(REAL));
259             memset(raySumFiltered,0,nelem*sizeof(REAL));
260
261             for(INTEGER nr = 0; nr < GeoPar.nrays; nr++)
262             {
263                 wray(np, nr, list, weight, &numb, &snorm);
264
265                 if(numb > 0)
266                 {
267                     val = 0.0;
268                     tsum = 0.0;
269                     // val = R*x -y
270                     //Iterate over all pixels of the trace
                            LINE ray
271                     //weight the lengths of the ray
                            segments
272
273                     for(INTEGER nb = 0; nb < numb; nb++)
274                     {
275                         tsum += x[list[nb]] * weight[nb];
276                     }
277
278                     val = tsum - Anglst.prdta(np,nr);
279
280                     raySum[nr]          = val;
281                     raySumFiltered[nr]  = val;
282                 }
283             }
284
285             //console1D(raySumFiltered, GeoPar.nrays, "
                    rayBefore");
286
287
288
289             snfft(raySumFiltered,n,1);
290
291             //consoleComplex(raySumFiltered, nelem, np, "
                    complex");
292
293
294
295             int cnt = 0;
296             // Apply the filter
297             for(int l=0;l<nelem;l+=2)
298             {
```

```
299                 //Apply filter on real and complex value
300                 raySumFiltered[l]   *= filterValues[cnt];
301                 raySumFiltered[l+1] *= filterValues[cnt];
302                 cnt++;
303             }
304
305             //Transform the values back into spatial space
306             snfft(raySumFiltered,n,-1);
307
308             //console1D(raySumFiltered, GeoPar.nrays, "
                   rayAfter");
309
310             //iterate over all x-rays again to write them
                   into the correct position
311             for(INTEGER nr = 0; nr < GeoPar.nrays; nr++)
312             {
313                 wray(np, nr, list, weight, &numb, &snorm);
314
315                 if(numb > 0)
316                 {
317                     for(INTEGER nb = 0; nb < numb; nb++)
318                     {
319                         unfiltered[list[nb]] += (weight[nb]
                               * raySum[nr]);
320                         filtered[list[nb]]   += (weight[nb]
                               * raySumFiltered[nr]);
321                     }
322                 }
323             }
324         } //finished all projections
325     }
326
327     if(n != NULL)
328     {
329         delete [] n;
330     }
331
332     if(raySum != NULL)
333     {
334         delete [] raySum;
335     }
336
337     if(raySumFiltered != NULL)
338     {
339         delete [] raySumFiltered;
340     }
341
342     if(filterValues != NULL)
343     {
344         delete [] filterValues;
345     }
346
347
348
349     return;
350 }
351
```

```
352 //
    ==========================================================================
353 // This function performs projection followed by
    backprojection for r=R'*R*p
354 void alp2_class::RtRp(REAL *r, INTEGER* list, REAL* weight,
     REAL *p)
355 {
356     REAL val;
357     INTEGER numb;
358     REAL snorm;
359
360     memset(r,0,GeoPar.area*sizeof(REAL));
361     if(!GeoPar.strip)
362     {
363         // LINE MODE
364         for(INTEGER np = 0; np < GeoPar.prjnum; np++)
365         {
366             for(INTEGER nr = 0; nr < GeoPar.nrays; nr++)
367             {
368                 wray(np, nr, list, weight, &numb, &snorm);
369
370                 if(numb > 0)
371                 {
372                     val = 0.0;
373
374                     // val = R*p
375                     for(INTEGER nb = 0; nb < numb; nb++)
376                     {
377                         val += p[list[nb]] * weight[nb];
378                     }
379
380                     // r = R'*R*p
381                     for(INTEGER nb = 0; nb < numb; nb++)
382                     {
383                         r[list[nb]] += weight[nb]*val;
384                     }
385                 }
386             }
387         }
388     }
389 }
390
391
392 //
    ==========================================================================
393 // init function
394 INTEGER alp2_class::Init()
395 {
396   /*f = fopen("recon.txt", "a");
397   if (f == NULL)
398   {
399       fprintf(output,"Error opening file!\n");
400       exit(1);
401   }
402   */
403
```

```
404
405   BOOLEAN eol;
406   //g = (REAL*)malloc(GeoPar.area*sizeof(REAL));
407   //p = (REAL*)malloc(GeoPar.area*sizeof(REAL));
408   //RRp = (REAL*)malloc(GeoPar.area*sizeof(REAL));
409   go  = new REAL [GeoPar.area];
410   g   = new REAL [GeoPar.area];
411   p   = new REAL [GeoPar.area];
412   RRp = new REAL [GeoPar.area];
413   tmp = new REAL [GeoPar.area];
414   z   = new REAL [GeoPar.area];
415
416
417   memset(p,0,GeoPar.area*sizeof(REAL));
418   memset(go,0,GeoPar.area*sizeof(REAL));
419
420   beta_type = InFile.getnum(TRUE, &eol);
421   if(eol)
422   {
423     fprintf(output, "\n       **** beta parameter not
        specified, assuming 0 ***");
424     beta_type = 0;
425   }
426
427   filter_type = InFile.getint(TRUE, &eol);
428   if(eol)
429   {
430     fprintf(output, "\n       **** filtertype parameter not
         specified, assuming 0 ***");
431     filter_type = 0;
432   }
433
434   if(filter_type == 1)
435   {
436     static const INTEGER windowFunctionTypes[4] =
437     {
438         CHAR2INT('r', 'a', 'm', 'p'),
439         CHAR2INT('h', 'a', 'm', 'm'),
440         CHAR2INT('t', 'r', 'i', 'a'),
441         CHAR2INT('c', 'o', 'm', 'b')
442     };
443
444     ramp = CHAR2INT('r', 'a', 'm', 'p');
445     hamm = CHAR2INT('h', 'a', 'm', 'm');
446     triangular = CHAR2INT('t', 'r', 'i', 'a');
447     combination = CHAR2INT('c', 'o', 'm', 'b');
448
449
450     windowFunction = InFile.getwrd(TRUE, &eol,
        windowFunctionTypes, 4);
451
452     rampDelta = 0.3;
453     hamm_alphaStd = 0.54;
454     triangular_LStd = 0.005;
455     comb_mu_default = 0.001;
456     comb_alpha_default = 0.54;
457
458
```

```
459     if(windowFunction == ramp)
460     {
461         rampDelta = InFile.getnum(FALSE, &eol);
462 
463         if(eol)
464         {
465             //Set sigma to std. value
466             rampDelta = stdRampDelta;
467         }
468         fprintf(output,"\n         **** Window Function is
              Ramp Function with sigma %.10f\n", rampDelta);
469     }
470     else if(windowFunction == hamm)
471     {
472         //Hamm with alpha = 0.54 is called Hanning Window
473         hamm_alpha = InFile.getnum(FALSE, &eol);
474 
475         if(eol)
476         {
477             hamm_alpha = hamm_alphaStd;
478         }
479 
480         fprintf(output,"\n         **** Window Function is
              Hamm Window with alpha %.10f\n",hamm_alpha);
481     }
482     else if(windowFunction == triangular)
483     {
484         //triangular Window Function with parameter add to
              L
485         triangular_L = InFile.getnum(FALSE, &eol);
486 
487         if(eol)
488         {
489             triangular_L = triangular_LStd;
490         }
491 
492         fprintf(output,"\n         **** Window Function is
              Triangular Window with L %.10f\n",triangular_L);
493     }
494     else if(windowFunction == combination)
495     {
496         //Combinational filter of ramp filter with hamming
              window
497         comb_mu = InFile.getnum(FALSE, &eol);
498         comb_alpha = InFile.getnum(FALSE, &eol);
499 
500         if(eol)
501         {
502             comb_mu = comb_mu_default;
503             comb_alpha = comb_alpha_default;
504         }
505 
506         fprintf(output,"\n         **** Window Function is
              Combination filter with mu %.5f and alpha %.5f\n
              ",comb_mu, comb_alpha);
507     }
508     else
509     {
```

```
510           windowFunction == ramp;
511           rampDelta = stdRampDelta;
512           fprintf(output,"\n       **** No Window function
                 specified, taking ramp with %.2f\n", rampDelta);
513       }
514     }
515       fprintf(output,"\n         **** Beta Parameter set to %f
             \n",beta_type);
516       fprintf(output,"\n         **** Filter Parameter set to
             %d\n",filter_type);
517
518       return 0;
519   }
520   //
      ================================================================

521   // reset function
522   INTEGER alp2_class::Reset()
523   {
524     if (go!=NULL) delete [] go;
525     if (g!=NULL)  delete [] g;
526     if (p!=NULL)  delete [] p;
527     if (tmp!=NULL)delete [] tmp;
528     if (RRp!=NULL)delete [] RRp;
529     if (z!=NULL)  delete [] z;
530
531     return 0;
532   }
533
534
535   void alp2_class::console1D(REAL* array1D, int length, std::
         string text)
536   {
537       if(text.length() !=0)
538       {
539           fprintf(f,"%s\n",text.c_str());
540       }
541
542       for(INTEGER i = 0; i < length; i++)
543       {
544           fprintf(f,"%.3f,", array1D[i]);
545       }
546
547       fprintf(f,"\n");
548
549   }
550
551   void alp2_class::consoleComplex(REAL* complexFFT, int
         length, int np, std::string text)
552   {
553       if(text.length() !=0)
554       {
555           fprintf(f,"%s\n",text.c_str());
556       }
557
558       //fprintf(f,"h=[");
559       for(int l=0;l<length;l+=2)
560       {
```

```
561             fprintf(f,"%.10f,",  complexFFT[l]);
562         }
563
564         fprintf(f,"\n");
565
566         for(int l=1;l<length;l+=2)
567         {
568             fprintf(f,"%.10f,",  complexFFT[l]);
569         }
570
571         fprintf(f,"\n");
572
573 }
574
575 int cnt =0;
576 //
    ==========================================================================

577 // Run
578 BOOLEAN alp2_class::Run(REAL* recon, INTEGER* list, REAL*
    weight, INTEGER iter)
579 {
580     cnt++;
581   // prjnum number of projections
582   // pinc spacing between the detectors
583   // nrays number of rays in a projection
584   // strip boolean, true if the ray spacing is uniform
585   // area nelem*nelem
586   // nelem number of elements in the reconstruction grid
587   // midpix is (nelem-1)/2
588   // picsiz length of the side of a pixel
589
590
591   //Computing gradient g = R'*R*x - R'*y
592
593     if(filter_type == 0)
594     {
595         //use marcelo's implementation without filter
596         grad_x(z,g,list,weight,recon);
597     }
598     else if(filter_type == 1)
599     {
600         //transform the data into the frequency and apply
              ramp filter
601         grad_x_filter(z, g,list,weight,recon);
602     }
603     else
604     {
605         fprintf(output,"filter_type not supported, breaking
              \n");
606         return TRUE;
607     }
608
609
610
611   //compute beta parameter of CG  (g'*R'*R*p)/(p'*R'*R*p)
612   if(iter > 1)
613   {
```

```
614       if(beta_type == 0 ){ beta_k = inner(z,RRp)/RptRp;}
615       if(beta_type == 1 ){ beta_k = inner(z,g)/inner(p,go);}
616     }
617     else
618     {
619       beta_k = 0;                    // first iteration is just
             gradient
620     }
621
622     //compute new conjugate direction p= -g+beta_k*p
623     comp_p(p,z,beta_k);
624
625
626     //compute step alpha  (-g'*p)/(p'*R'*R*p)
627       //RRp = Ap
628       RtRp(RRp,list,weight,p);
629       //RRp = pk
630       //inner is a sum value
631       RptRp=inner(p,RRp);
632       //g = gk
633       alpha_k = - inner(g,p)/RptRp;
634
635     //console1D(g, GeoPar.nrays, "g");
636     //console1D(z, GeoPar.nrays, "z");
637
638
639     //compute new solution recon = recon + alpha_k*p
640     comp_x(recon,p,alpha_k);
641     //copy old gradient go = -g
642     comp_p(go,g,0);
643
644     return FALSE;
645 }
```