

Copyright
by
Tharit Tangkijwanichakul
2021

The Thesis committee for Tharit Tangkijwanichakul
Certificates that this is the approved version of the following thesis:

**Chain of Operators for Inverse Hessian Estimation in
Least-squares Migration**

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Sergey Fomel, Supervisor

Dr. Stephen Grand

Dr. Kyle Spikes

**Chain of Operators for Inverse Hessian Estimation in
Least-squares Migration**

by

Tharit Tangkijwanichakul

THESIS

Presented to the Faculty of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

BACHELOR OF SCIENCE WITH HONORS

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2021

Acknowledgments

For my past 4 years at the University of Texas at Austin as an undergraduate student, I have received many supports from people I encountered. First, I would like to thank Dr. Sergey Fomel for his role as both academic advisor and exemplary scientist. I benevolently have been influenced by him in many ways. I am thankful for his style of not hand-holding his students. His guidance is always useful but always leaves the room for me to think and self-study to finally come up with conclusions by myself. Most importantly, it is his emphasis on reproducibility of research which encourages the intellectual honesty and transparency. That being said, for working with him, not only I have sharpen my technical expertise but I also have a chance to refine my character.

I also want to thank many professors in Jackson School of Geosciences whom I have encountered. I particularly thank Dr. Kyle Spikes and Thomas Hess who acted as my very first research supervisors and introduced me the field of seismic processing. This led me to discover my interest and eventually meet Dr. Sergey Fomel. I also want to mention Dr. Stephen Grand, Dr. Clark Wilson, and Dr. Luc Lavier for their help in building my foundation in geophysics.

I am grateful to many graduate students and research scientists I have met along the way: Harpreet Kaur, Ben Gremillion, Nam Pham, Zhicheng Geng, Luke Decker, Rebecca Gao, Hector Corzo, Dr. Yuzhi Shi, Dr. Ray Abma. Interacting with them helps me grow intellectually and personally. Prior to working with the Texas Consortium for Computational Seismology (TCCS), I used to be rather overconfident

student. However, having met and got to know both present and former TCCS students, I became more humble. I realize that there are entirely different playing field out there. This in turn aspired me to never stop learning.

Last but not least, I want to thank my soon-to-be colleague at PTTEP and my family back in Thailand for supporting me for all these years in the United States. Even without their physical presence, I never feel walking alone.

THARIT TANGKIJWANICHAKUL

The University of Texas at Austin

April 2021

Chain of Operators for Inverse Hessian Estimation in Least-squares Migration

Tharit Tangkijwanichakul, B.S.

The University of Texas at Austin, 2021

Supervisor: Dr. Sergey Fomel

I propose a novel way to approximate the inverse Hessian operator by a chain of weights in space and frequency domains. I call this method the Chain of Operators. Fundamentally, we approximate the complex operator (Hessian) via the chain of elementary operators (weights). This method is physically intuitive and provides a simple way to invert for the inverse Hessian. The method can be applied either for compensating migrated images i.e. applying approximated Inverse Hessian directly to the migrated image or in the form of a preconditioner inside iterative least-squares reverse-time migration (LSRTM).

Tests on synthetic and real data show that this approach provides an effective approximation of the Inverse Hessian while having the minimal cost of forward and inverse FFTs (Fast Fourier Transforms). However, the effectiveness of the methods varies quite greatly from dataset to dataset.

When used for compensating migrated images, the method proves noticeably effective in the synthetic Marmousi dataset. With a real Viking Graben dataset, the

method has difficulties in estimating weights due to muted area of the data. However, with some adjustments, the method noticeably improves the image resolution and amplitude balance. Plus, when examining the frequency spectrum, compensating the image by an approximated inverse Hessian emulates the effect of iterative LSRTM.

When used the approximated Inverse Hessian form a preconditioner in LSRTM, the result with the Marmousi dataset shows that it can significantly accelerate the convergence of LSRTM and achieves high-quality imaging results in fewer iterations. However, when applied to the Sigsbee model, the method is only marginally effective. Particularly. it appears to have difficulties in solving for the weight in the frequency domain as migrating the Sigsbee data with RTM tends to produce stronger low-frequency noise.

Overall, my experiments can be considered as a proof of concept that we can approximate the complex operator via the chain of elementary operators in the particular application to improving the efficiency of Least-squares Migration.

Table of Contents

Acknowledgments	iv
Abstract	vi
List of Tables	ix
List of Figures	x
Chapter 1. Introduction	1
Chapter 2. Chain of operator: Theory	7
Chapter 3. Chain of operators: Experiments	13
Chapter 4. Conclusions	33
Appendix	35
Bibliography	48
Vita	51

List of Tables

List of Figures

1.1	Simple Layer - Shot record	2
1.2	Simple Layer - RTM Image(left) without Inverse Hessian (right) with Inverse Hessian	3
2.1	Schematic drawing of chain of operators	7
3.1	(a) Marmousi - Shot gathers (b) Migration velocity for prestack migration	13
3.2	Marmousi - RTM Image	14
3.3	Marmousi - \mathbf{W}^{-1} 2, 5, 10 iteration of update	15
3.4	Marmousi - \mathbf{W}_f^{-1} 2, 5, 10 iteration of update	16
3.5	Marmousi - Poststack Deconvolution Image 2, 5, 10 iteration of update	18
3.6	Viking - (a) Zero-offset shot (b) Dix Velocity for Migration	19
3.7	Viking - (left) standard Zero-offset migration (mid) Deconvolved image by chain (right) Iterative LSM	19
3.8	Viking - Zoom-in comparison	20
3.9	Viking - Frequency spectrum of - blue: standard ZORTM, red: Deconvolved by chain, pink: Iterative LSM	21
3.10	Changes in Conjugate-gradient algorithms to incorporate preconditioner	22
3.11	Marmousi - (a) LSRTM Image without and (b) with Preconditioner after 20 iterations	23
3.12	Marmousi - Frequency spectrum of migrated image - blue: standard RTM, red: LSRTM, pink: Precondition LSRTM	23
3.13	Marmousi - Zoom-in comparison 1 between CG and Preconditioned CG	24
3.14	Marmousi - Zoom-in comparison 2 comparison between CG and Preconditioned CG	24
3.15	Marmousi - Zoom-in comparison 3 comparison between CG and Preconditioned CG	24
3.16	Marmousi - Zoom-in comparison 4 comparison between CG and Preconditioned CG	25

3.17 Marmousi - Normalize data misfit dash=without preconditioner solid(green)=with weight in space only solid(purple)=with chain preconditioner	25
3.18 Marmousi - (left)LSRTM Image without Precondtitioner (mid) with Preconditioner (\mathbf{W}, \mathbf{W}_f) (right) Preconditioner (\mathbf{W} only)after 100 iterations	26
3.19 Sigsbee - Stratigraphic velocity	27
3.20 Sigsbee - Synthetic shots	28
3.21 Sigsbee - RTM image (a) before remove low-frequency noise (b) after remove low-frequency noise	29
3.22 Sigsbee - (a) Weight in space domain \mathbf{W} (b) in Frequency domain \mathbf{W}_f	29
3.23 Sigsbee - LSRTM image without preconditioner	30
3.24 Sigsbee - LSRTM image with preconditioner	30
3.25 Sigsbee - LSRTM image without preconditioner after remove low frequency	31
3.26 Sigsbee - LSRTM image with preconditioner after remove low frequency	32

Chapter 1

Introduction

REVIEW OF SEISMIC IMAGING

Seismic Migration as an adjoint operator

The seismic reflection data \mathbf{d} that is acquired in the survey has an approximately linear relationship with the earth reflectivity model \mathbf{m} as $\mathbf{d} = \mathbf{Lm}$ (Tarantola, 1984b; Wiggins, 1972). The contrast of the lithology in the subsurface leads to the difference in acoustic impedance and hence reflectivity \mathbf{m} . \mathbf{L} is the operator that incorporates the physics of seismic wave propagation involved from the source to the subsurface and back to the receivers. In practice, choices of \mathbf{L} can be classified into 2 categories: ray-based and wave-equation-based (Jones, 2014). The velocity model of the subsurface embedded in \mathbf{L} is the crucial parameter that controls the structural accuracy of the migrated image. In the subsequent work presented in this thesis, I am dealing exclusively with the situation (i.e the stage of imaging project) where the velocity model is assumed to be accurately estimated and left unchanged throughout. Otherwise, the problem will be non-linear which requires the different optimization approach (Pica et al., 1990).

The task of seismic imaging (or migration) is to construct the earth image represented by \mathbf{m} from the acquired data \mathbf{d} . It is recognized that seismic migration is an adjoint operator \mathbf{L}^T of its associated forward modeling \mathbf{L} (Claerbout, 1992). In other words, the seismic image is constructed by back-projecting the seismic energy

acquired on the surface to the point in the subsurface where that energy comes from. This process is kinematically equivalent to applying adjoint operator \mathbf{L}^T to the data \mathbf{d} .

However, one can see that $\mathbf{m}_{\text{mig}} = \mathbf{L}^T \mathbf{d}$ is not the solution to the problem $\mathbf{d} = \mathbf{Lm}$. In fact, the least-squares solution is $\mathbf{m} = (\mathbf{L}^T \mathbf{L})^{-1} \mathbf{L}^T \mathbf{d}$. The missing term in \mathbf{m}_{mig} is $(\mathbf{L}^T \mathbf{L})^{-1}$ which is called Inverse Hessian. In other words, the migrated image is when I assume the inverse Hessian is identity matrix which is not generally true.

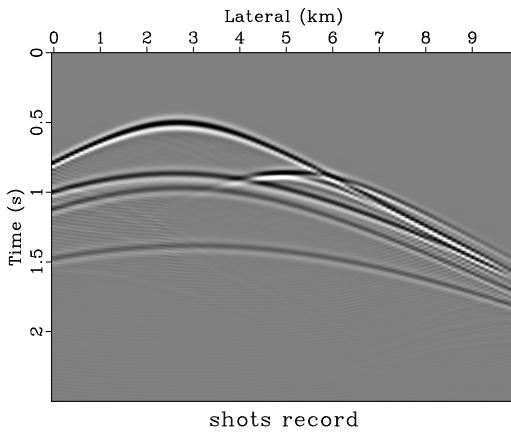


Figure 1.1: Simple Layer - Shot record chapter-intro/..../chapter-intro/intro shots

To illustrate the effect of incorporating the inverse Hessian term, consider the synthetic shot record \mathbf{d} in Figure 1.1 generated using finite-difference wave propagator. If I apply the migration operator \mathbf{L}^T , I get the image in Figure 1.2(a). In contrast, the image in Figure 1.2(b) is when I compensated the standard migrated image $\mathbf{m}_{\text{mig}} = \mathbf{L}^T \mathbf{d}$ with the Inverse Hessian $(\mathbf{L}^T \mathbf{L})^{-1}$ through the iterative inversion. One can see that both images are structurally similar. This is because it is only the velocity model that controls the structural accuracy of the migrated image. Here, I do not change or make an update to the velocity model through the iterative inversion.

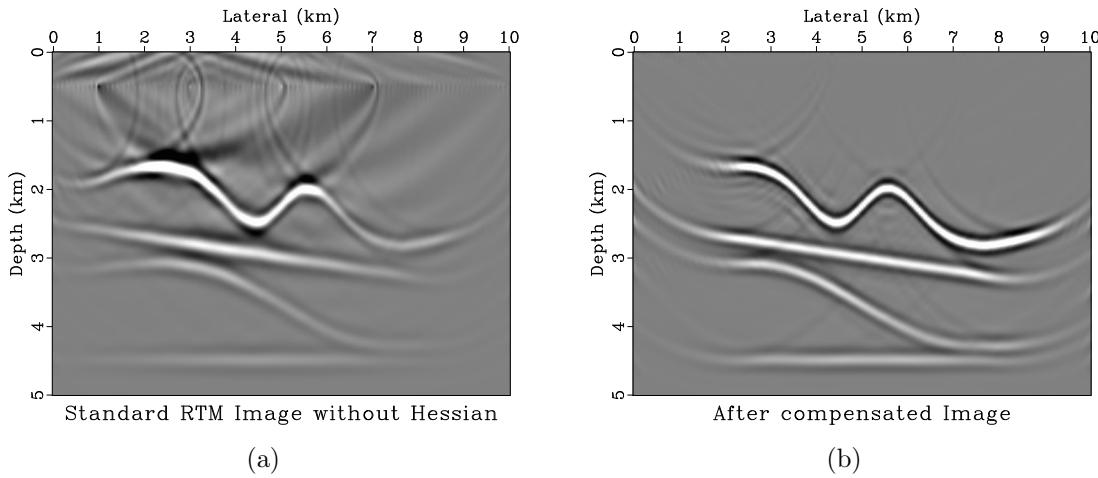


Figure 1.2: Simple Layer - RTM Image(left) without Inverse Hessian (right) with Inverse Hessian chapter-intro/..../chapter-intro/intro mig1,lsrtm

One can notice obvious differences in terms of amplitude balancing and resolutions. The $\mathbf{m} = (\mathbf{L}^T \mathbf{L})^{-1} \mathbf{L}^T \mathbf{d}$ image has more balancing amplitude and improved resolution. This is the simple illustration that having accurately estimated inverse Hessian is beneficial.

DEVELOPMENT IN INVERSE HESSIAN ESTIMATION

Least-squares migration (LSM) is a well-established technique for improving the quality of seismic imaging through inversion (Nemeth et al., 1999; Ronen and Liner, 2000). By formulating seismic imaging as a linear estimation problem, least-squares migration utilizes the power of iterative inversion for recovering the reflectivity of subsurface structures. In recent years, least-squares imaging, in particular LSRTM (least-square reverse-time migration) has found many successful applications (Dai et al., 2012; Wang et al., 2016; Wong et al., 2011).

The least-squares formulation involves the inverse Hessian operator. As recognized by previous researchers, the algorithm can be accelerated if the inverse Hessian matrix can be accurately approximated. Normally, the inverse Hessian is replaced by the identity matrix i.e. when only migration is performed, which is not a correct assumption. This is because it is not practical to form the inverse Hessian explicitly as it involves directly inverting an extremely large matrix.

In the conventional approach, the inverse Hessian is approximated by iterative methods, such as conjugate gradients (Tarantola, 1984a; Sun et al., 2016; Xue et al., 2016). This raises the cost of LSRTM to the cost of migration and modeling multiplied by the number of iterations. Only a small number of iterations can be affordable in practice.

To reduce this cost or, alternatively, to reduce the number of iterations by accelerating the iterative convergence, a number of methods have been proposed in the literature for approximating the inverse Hessian using relatively inexpensive computations. Rickett (2003) estimated a simple scaling operator to approximate the inverse Hessian with a diagonal matrix. Guitton (2004) and Greer et al. (2018) extended this approach to matching filters, approximating the inverse Hessian with a nonstationary convolution operator. This approach is also known as *migration deconvolution* (Hu et al., 2001; Yu et al., 2006). Aoki and Schuster (2009) approximated inverse Hessian as a deblurring filter for regularization and precondition scheme. Kaur et al. (2020) adopt the deep learning technique and use generative adversarial networks in a conditional setting (CycleGANs) to approximate inverse Hessian.

This work is motivated mainly by an asymptotic theory (Miller et al., 1987; Bleistein, 1987). The theory shows that the inverse Hessian can be represented as

a combination of weights in the time domain and the frequency domain. The result extends to the case of LSRTM (Hou and Symes, 2015, 2016).

TECHNICAL CONTRIBUTIONS

I propose to approximate the inverse Hessian by reformulating by simultaneously estimating a chain of weights in the space and frequency domains from initial images. I design this chain to be symmetric to preserve the symmetric property of the Hessian. Our method has a relatively low computational cost compared to the overall Least-squares migration process. Once the weight matrices are estimated, they can efficiently approximate the inverse Hessian. Alternatively, the approximate inverse Hessian can be incorporated in the form of a preconditioner to accelerate the convergence of iterative LSRTM.

THESIS OUTLINE

This undergraduate thesis organized according to the following outline:

- In Chapter 2, I first present the theory of chain of operators, problem formulation, and an estimation algorithm. In particular, I present the estimation of inverse Hessian applied to the Least-square Migration problem.
- In Chapter 3, I test the accuracy of the proposed approach using zero-offset synthetic data. I start with the synthetic data from the Marmousi model (Versteeg, 1994). Then, I also test with the real data from Viking Graben.
- Subsequently in Chapter 3, I use prestack Marmousi and Sigbee2A data to evaluate the effectiveness of the proposed approximation of Inverse Hessian by

using it as a preconditioner for LSRTM.

Chapter 2

Chain of operator: Theory

THEORY

Motivation of chain of operators

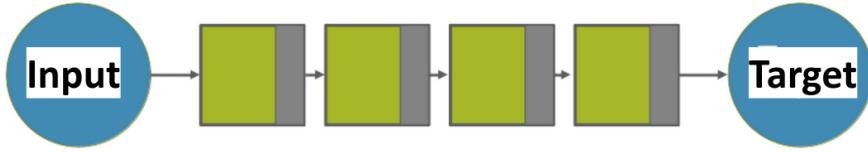


Figure 2.1: Schematic drawing of chain of operators
chapter-background/..//chapter-background/bg chain

The motivation behind the idea of the chain of operators is that one can approximate the complex operator through the chain of parameterized elementary operators. As illustrated in Figure 2.1, the input source is related to the target through the chain of parameterized elementary operators (green). Between each of these elementary operators are predetermined function (grey) chosen appropriately for the domain of application. In this work as motivated by the asymptotic theory mentioned earlier, this predetermined function is a Fourier (and Inverse Fourier) Transform.

Problem formulation

Given surface seismic data \mathbf{d} , the first migrated image \mathbf{m}_1 is obtained by applying seismic migration as the adjoint of forward linear modeling operator \mathbf{L} :

$$\mathbf{m}_1 = \mathbf{L}^T \mathbf{d} . \quad (2.1)$$

Next, we can model and migrate again, generating the second image

$$\mathbf{m}_2 = \mathbf{L}^T \mathbf{L} \mathbf{m}_1 \quad (2.2)$$

If we can find an effective approximation for the Hessian operator from equation (2.2)

$$\mathbf{H} \approx \mathbf{L}^T \mathbf{L} \quad (2.3)$$

and its inverse, then applying this inverse to the initial image \mathbf{m}_1 will provide an effective approximation for the desired least-squares image

$$\mathbf{m}_3 = \mathbf{H}^{-1} \mathbf{m}_1 \approx (\mathbf{L}^T \mathbf{L})^{-1} \mathbf{L}^T \mathbf{d} . \quad (2.4)$$

As discussed earlier that we cannot afford the cost of directly inverting the matrix, it is clear that the desired form of approximated Hessian should be easy, preferably trivial, to invert.

Chain operator in Least-squares Migration

We propose to approximate the connection between \mathbf{m}_1 and \mathbf{m}_2 in equation (2.2) using a chain of weights alternating between the original domain and the Fourier domain, as follows:

$$\mathbf{m}_2 \approx \mathbf{W} \mathbf{F}^{-1} \mathbf{W}_f \mathbf{F} \mathbf{W} \mathbf{m}_1 , \quad (2.5)$$

where \mathbf{W} and \mathbf{W}_f are diagonal matrices, and \mathbf{F} represents the Fourier transform. \mathbf{W} represents the weighting factor that matches the amplitude of the \mathbf{m}_1 and that of \mathbf{m}_2 in the original domain of the image (time or depth). \mathbf{W}_f is a weighting factor (or filter) that matches the frequency of those two corresponding images. Our goal is to estimate \mathbf{W} and \mathbf{W}_f from the given pairs of \mathbf{m}_1 and \mathbf{m}_2 ,

In the digital filtering terminology, the operation of $\mathbf{F}^{-1}\mathbf{W}_f\mathbf{F}$ is equivalent to convolution. Therefore, the proposed chain represents a combination of scaling and convolution in a symmetric setting. The symmetry is important because the Hessian operator ($\mathbf{L}^T\mathbf{L}$), which we are approximating using this representation, is symmetric by definition.

Once we obtain \mathbf{W} and \mathbf{W}_f , inverting the chain is straightforward as each weight matrices is a diagonal matrix. The compensated migrated image \mathbf{m}_3 from equation (2.4) becomes

$$\mathbf{m}_3 = \mathbf{W}^{-1}\mathbf{F}^{-1}\mathbf{W}_f^{-1}\mathbf{F}\mathbf{W}^{-1}\mathbf{m}_1. \quad (2.6)$$

Solving for \mathbf{W} and \mathbf{W}_f

To solve the nonlinear representation of chain of operators in equation (2.5), we introduce intermediate vectors \mathbf{x}_1 and \mathbf{x}_2 and treat them as additional unknown variables, splitting the initial condition into three linear equations, as follows:

$$\begin{aligned} \mathbf{x}_1 &\approx \mathbf{W}\mathbf{m}_1 \\ \mathbf{x}_2 &\approx \mathbf{F}^{-1}\mathbf{W}_f\mathbf{F}\mathbf{x}_1 \\ \mathbf{m}_2 &\approx \mathbf{W}\mathbf{x}_2 \end{aligned} \quad (2.7)$$

In the vector notation,

$$\mathbf{r} = \begin{bmatrix} \mathbf{W} & -\mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}^{-1}\mathbf{W}_f\mathbf{F} & -\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{W} & -\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{m}_1 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{m}_2 \end{bmatrix} \approx \mathbf{0}, \quad (2.8)$$

and the solution is found by minimizing the residual vector \mathbf{r} while updating \mathbf{w} , \mathbf{w}_f , \mathbf{x}_1 , and \mathbf{x}_2 . Linearizing equation (2.8) with respect to perturbations in unknown variables, we arrive at

$$\begin{bmatrix} -\mathbf{I} & \mathbf{0} & \text{diag}[\mathbf{m}_1] & \mathbf{0} \\ \mathbf{F}^{-1}\mathbf{W}_f\mathbf{F} & -\mathbf{I} & \mathbf{0} & \mathbf{F}^{-1}\text{diag}[\mathbf{F}\mathbf{x}_1] \\ \mathbf{0} & \mathbf{W} & \text{diag}[\mathbf{x}_2] & \mathbf{0} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_1 \\ \Delta\mathbf{x}_2 \\ \Delta\mathbf{W} \\ \Delta\mathbf{W}_f \end{bmatrix} \approx -\mathbf{r} \quad (2.9)$$

We solve (2.9) using the conjugate-gradient method with shaping regularization that enforces the smoothness of estimated variables (Fomel, 2007b), then update \mathbf{W} , \mathbf{W}_f , and recalculate residual \mathbf{r} in (2.8) iteratively until convergence. This corresponds to a regularized Gauss-Newton approach for solving the original nonlinear system.

Noted that to choose the step size in the updating scheme, we use a simple binary search of step size. After obtaining the update vector from (2.9), we update the variables using step size = 1.0. Then, we check if the residual decreases or not. If it does not, we roll back the variables and update them using step size = 0.5. We repeat this fashion until the residual decreases. See the Appendix for complete code implementation (chain2dfft.c and Mchain2dfft.c)

It is clear that solving (2.5) is not guaranteed to achieve global minimum as the equation is non-convex with respect to \mathbf{W} and \mathbf{W}_f . To mitigate the problem of being subjected to local minima, the initial guess \mathbf{W}_0 and \mathbf{W}_f0 are supplied to the algorithms. To initialize the initial time/space weight \mathbf{W} , we use a smooth division

of \mathbf{m}_2 by \mathbf{m}_1 (Fomel, 2007a) and then take a square root. We set the initial frequency weight \mathbf{W}_f to 1.0.

Preconditioning with chain weights

To go beyond applying the inverse Hessian in Post-stack image, we propose to apply the weights \mathbf{W} and \mathbf{W}_f to form a preconditioner to speed up the convergence of iterative least-square migration. The original problem is to solve for \mathbf{m} where $\mathbf{d} \approx \mathbf{Lm}$. We apply the change of variables from \mathbf{m} to \mathbf{y} according to $\mathbf{m} = \mathbf{Py}$ where \mathbf{P} is the preconditioning matrix. Then the problem becomes

$$\mathbf{d} \approx \mathbf{LPy} \quad (2.10)$$

We then solve this equation for \mathbf{y} with the least-square objective function

$$J(\mathbf{y}) = \frac{1}{2} \|(\mathbf{LP})\mathbf{y} - \mathbf{d}\|_2^2, \quad (2.11)$$

which corresponds to the analytical solution

$$\mathbf{y} = (\mathbf{P}^T \mathbf{L}^T \mathbf{LP})^{-1} \mathbf{P}^T \mathbf{L}^T \mathbf{d}. \quad (2.12)$$

After iteratively inverting for \mathbf{y} , we recover the original variable $\mathbf{m} = \mathbf{Py}$.

The iterative inversion of finding \mathbf{y} is accelerated because according to the approximation of Hessian operator $\mathbf{L}^T \mathbf{L} \approx \mathbf{WF}^{-1} \mathbf{W}_f \mathbf{F} \mathbf{W}$

$$\begin{aligned} \mathbf{L}^T \mathbf{L} &\approx \mathbf{WF}^{-1} \mathbf{W}_f \mathbf{F} \mathbf{W} \\ &= (\mathbf{P} \mathbf{P}^T)^{-1}, \end{aligned} \quad (2.13)$$

the preconditioner \mathbf{P} can be constructed from chain weights as

$$\mathbf{P} = \mathbf{W}^{-1} \mathbf{F}^{-1} \mathbf{W}_f^{-1/2} \mathbf{F}. \quad (2.14)$$

This makes the inverted operator in equation (2.12) close to unitary,

$$(\mathbf{LP})^T \mathbf{LP} \approx \mathbf{I}, \quad (2.15)$$

which accelerates the convergence when the inversion is carried out by an iterative method.

Chapter 3

Chain of operators: Experiments

ZERO-OFFSET EXPERIMENTS

Chain as Migration Deconvolution Filter: Synthetic Marmousi data

We test the idea of the chain of operators with Marmousi data (Versteeg, 1994). The shot gathers volume is shown in Figure 3.1(a) and consists of 45 shots. The migration velocity is shown in Figure 3.1(b) which is obtained from smoothing the stratigraphic slowness. Figure 3.2 shows a reverse-time migrated image using a finite-difference wave propagator.

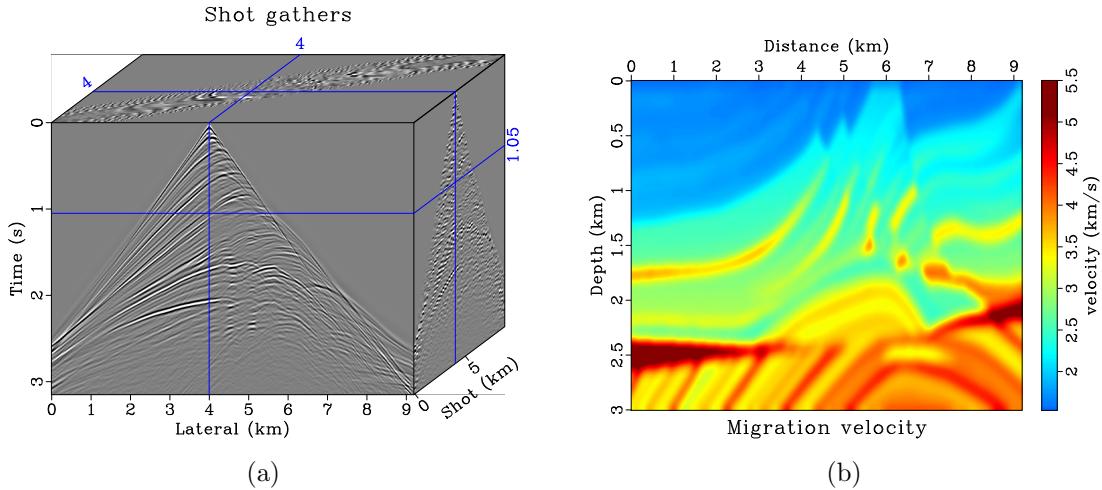


Figure 3.1: (a) Marmousi - Shot gathers (b) Migration velocity for prestack migration
chapter-lsrtm/.../chapter-lsrtm/pre mmbshots45,velmig

This work EAGE CITE was done under the supervision of Dr. Sergey Fomel.

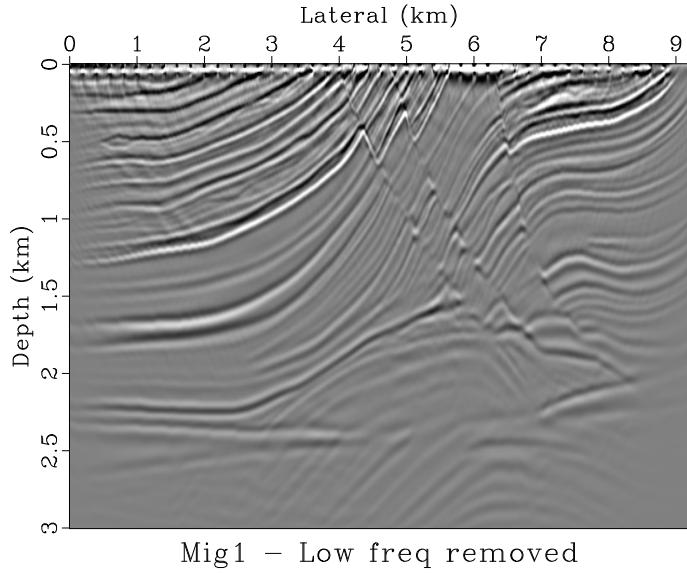


Figure 3.2: Marmousi - RTM Image [chapter-lsrtm/..//chapter-lsrtm/pre mmbmig1]

After we obtain the second migrated image \mathbf{m}_2 by remodeling and remigration exercise ($\mathbf{L}^T \mathbf{L}$) to \mathbf{m}_1 , we form the initial weights \mathbf{W}_0 by taking the square-root of smooth division $\frac{\mathbf{m}_2}{\mathbf{m}_1}$ as mentioned earlier.

We run the chain solver to estimate \mathbf{W} and \mathbf{W}_f for different numbers of iterations. The residual is about 1.9 %, 0.23 %, 0.21 % of the zeroth iteration (i.e. with $\mathbf{W} = \mathbf{W}_0$ and $\mathbf{W}_f = \mathbf{I}$) after 2, 5, 10 iterations respectively. These weights are shown in Figure 3.3 and 3.4. Notice that the weights in space domain \mathbf{W}^{-1} does not change much after more iterations, contrasting to the behavior of the weight in frequency domain \mathbf{W}_f^{-1}

Subsequently, we used chain weights to perform post-stack deconvolution as prescribed in equation (2.6). The result (Figure 3.5) shows an immediate improvement in resolution over that of the initial RTM image. The deconvolved image with 5 iterations has a higher resolution compared to the one run with 2 iterations. However,

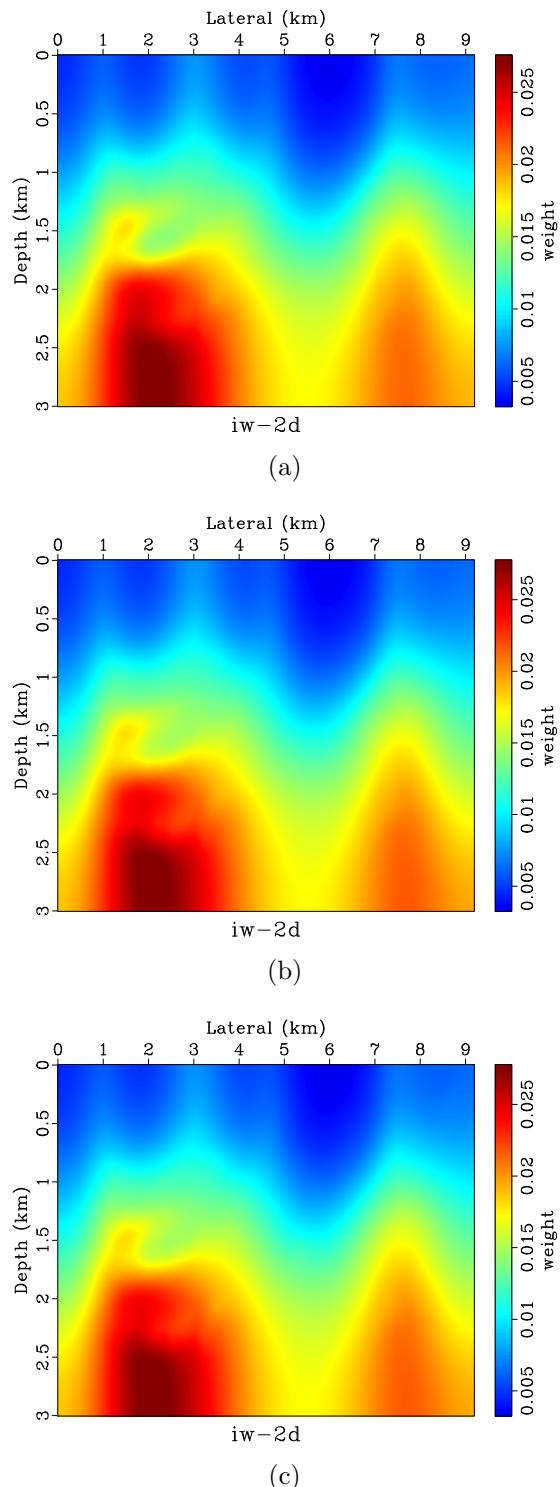


Figure 3.3: Marmousi - \mathbf{W}^{-1} 2, 5, 10 iteration of update
 chapter-lsrtm/..../chapter-lsrtm/pre iw,iw5,iw10

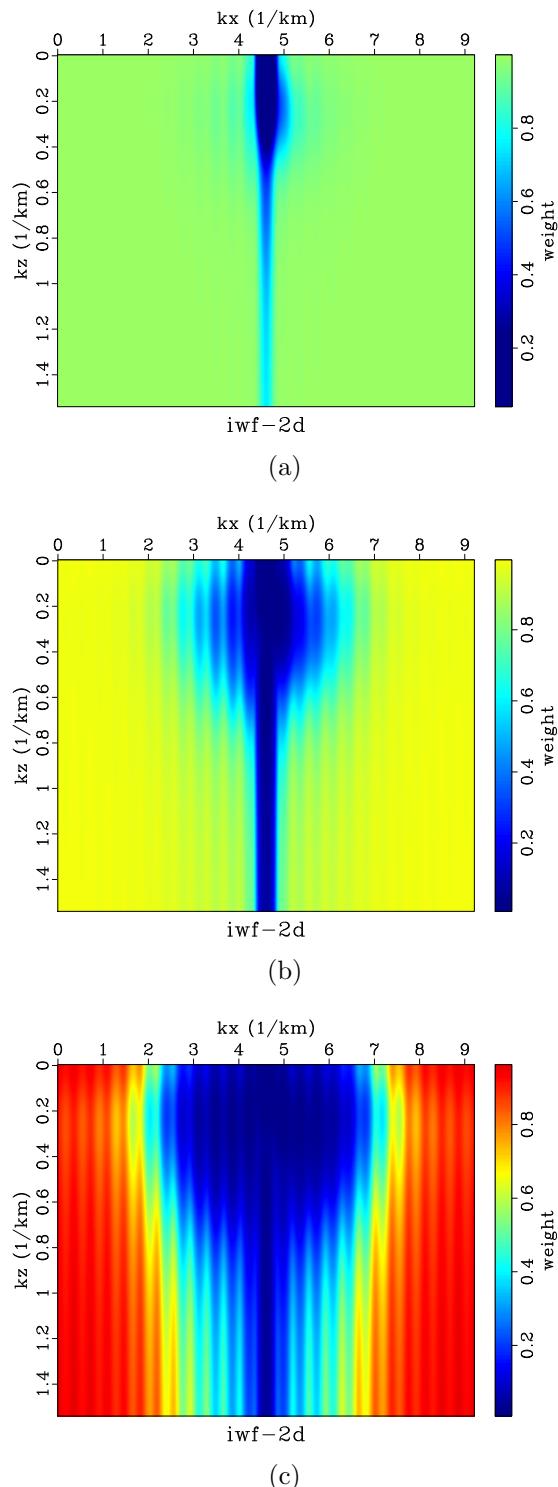


Figure 3.4: Marmousi - \mathbf{W}_f^{-1} 2, 5, 10 iteration of update
 chapter-lsrtm/..../chapter-lsrtm/pre iwf,iwf5,iwf10

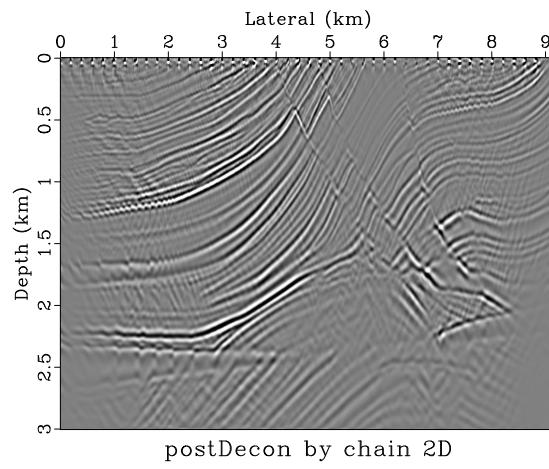
the image with 5 iterations gives a smoother image. This may be from the fact that we get smoother frequency weight \mathbf{Wf}^{-1} in the area where data mostly reside in Fourier domain i.e. 90° dip in (k_x, k_z) space.

Chain as a Deconvolution Filter: Real Viking Graben data

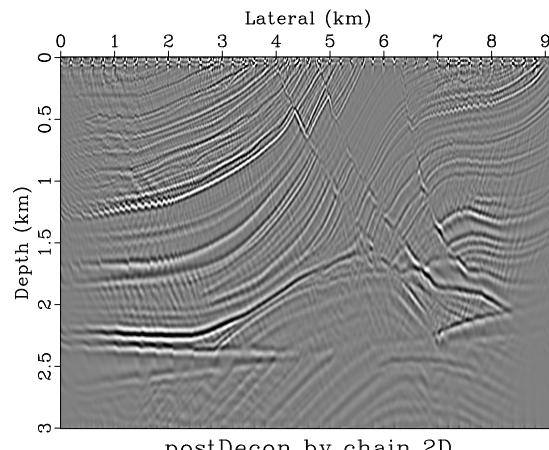
In this experiment, we compare the result of using the chain of operators as a deconvolution filter applied to a zero-offset migrated image as in equation (2.6) with the traditional zero-offset iterative least-square migration. The zero-offset data (Fig 3.6(a)) comes from the Viking Graben Field (Keys and Foster, 1998). Its corresponding migration velocity is in (3.6(b)). We migrated the data using zero-offset RTM.

Noted that in the zero-offset data, we have an area that we muted during the pre-processing (the area above 0.4s) Thus, the migrated image will have muted area in space domain near the surface too. The experiment shows that if we solve for \mathbf{W} , \mathbf{Wf} using a whole image, the result after applied deconvolution operator will have high-frequency artifacts in the area where the data was muted originally.

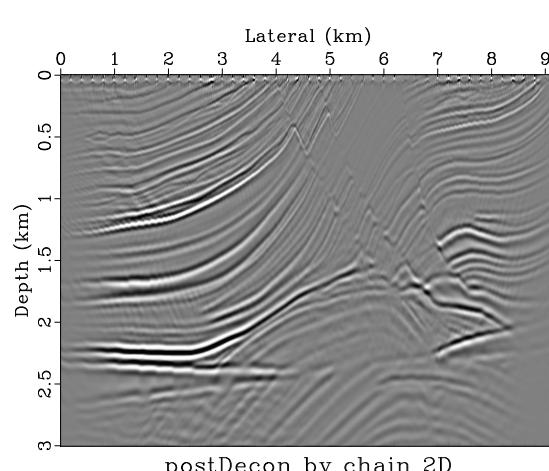
Hence, we will test the effectiveness of the deconvolution filter only in the area where we have the complete image. In other words, the weights are solved using the input two windowed stacks of the first and second migrated image. In Fig 3.7(a), 3.7(b), 3.7(c) we show windowed section of the migrated image, deconvolved image by chain weights, and traditional iterative least-square migration respectively. To be clear, the image from iterative LS comes from using the whole image as an input still. We can see the amplitude range of a deconvolutioned image and LSM image are about the same while that of a standard migrated image is in a different range.



(a)



(b)



(c)

Figure 3.5: Marmousi - Poststack Deconvolution Image 2, 5, 10 iteration of update
chapter-lsrtm/..../chapter-lsrtm/pre decon2,decon5,decon10

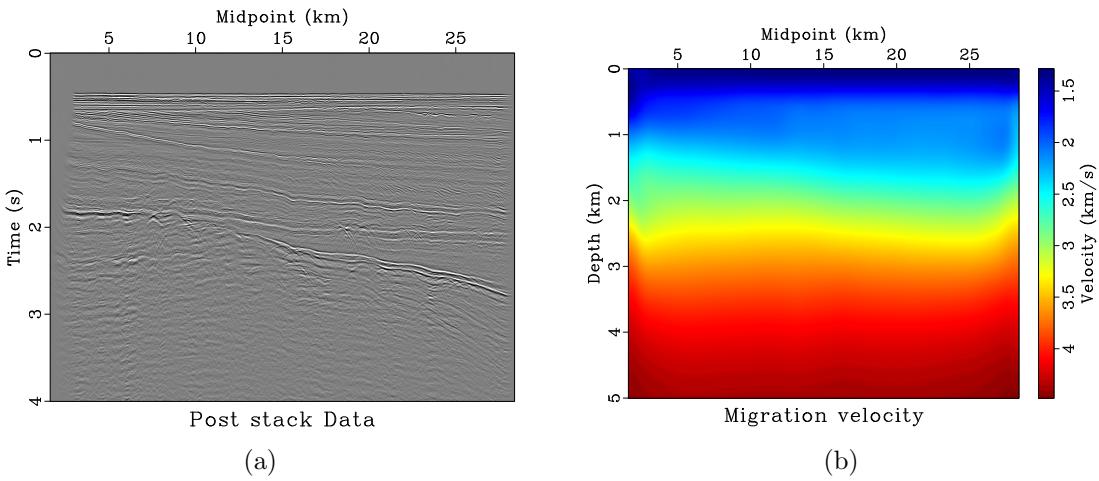


Figure 3.6: Viking - (a) Zero-offset shot (b) Dix Velocity for Migration
 chapter-lsrtm/..../chapter-lsrtm/vk zodata,veldix

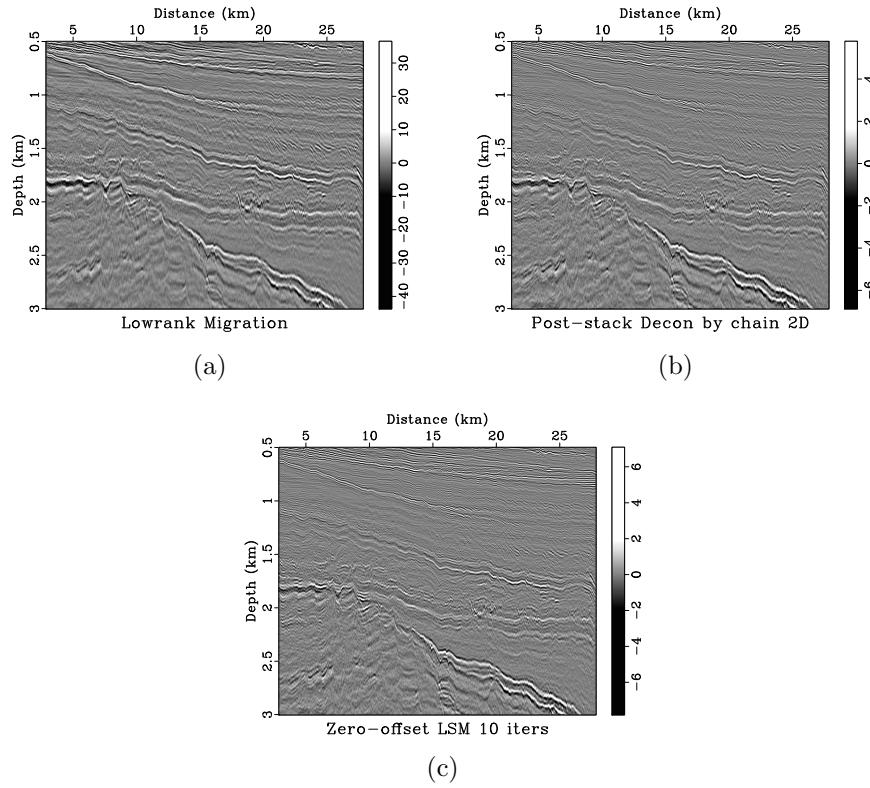


Figure 3.7: Viking - (left) standard Zero-offset migration (mid) Deconvolved image by chain (right) Iterative LSM
 chapter-lsrtm/..../chapter-lsrtm/vk cmig1,cdecon,lsm0

Fig 3.8 shows the zoom-in portion of these images. We can see that the layers in the deconvolutioned image look more similar to the LSM image in terms of amplitude balancing.

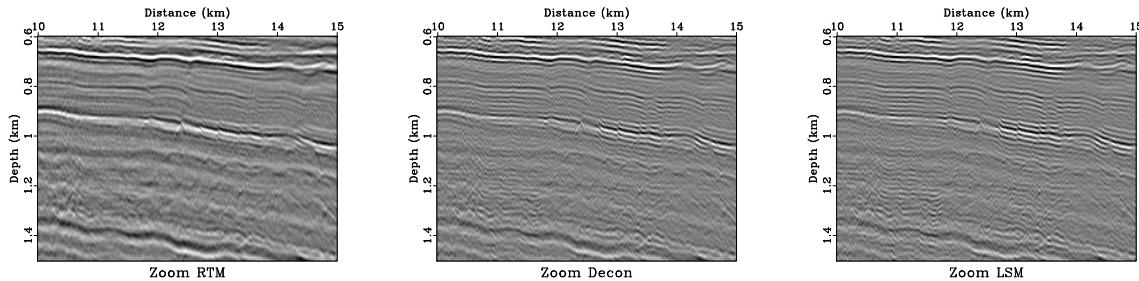


Figure 3.8: Viking - Zoom-in comparison chapter-lsrtm/..../chapter-lsrtm/vk zooms

In Fig 3.9, we use the average frequency spectrum as one of the available quantitative tools to compare the image resolution. Deconvolution by chain can recover high frequency but not as good as iterative LSM, which has a higher computational cost. The pattern of the frequency spectrum of deconvolutioned image by chain and that of iterative LSM appear similar which suggests that in this case, the chain deconvolution filter derived from the chain emulates the inverse Hessian estimated by iterative LSM.

So far, estimating the inverse Hessian operator by the chain of operators shows a promising result on the synthetic data. On the real Viking Graben data, we still gain the improvement of the image from using the method. However, the experiment shows that imperfections of the real data can limit the usability of the method.

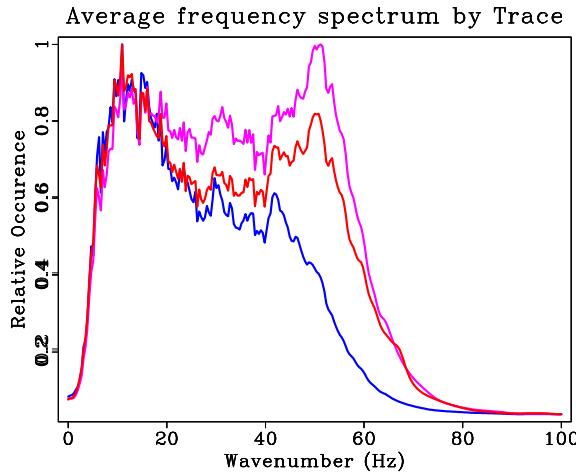


Figure 3.9: Viking - Frequency spectrum of - blue: standard ZORTM, red: Deconvolved by chain, pink: Iterative LSM

PRE-STACK EXPERIMENTS

Chain as a Preconditioner for Least-square RTM

We use the weights in Figure 3.3 and 3.4) to form a preconditioner according to equation (2.14). In particular, we chose the weights obtained after 2 update iterations. This choice was not made based on any preference. Rather, it was due to the availability of results at that time when this subsequent experiment was performed.

In Figure 3.10, we show the change made to the generic conjugate-gradient algorithms to incorporate the preconditioning matrix \mathbf{P} according to equation 2.14. The operator \mathbf{L} and \mathbf{L}^T can be any modeling/migration pair. In this case, we use Reverse-Time Migration (RTM). The illustration is modified from (Fomel et al., 2013). Here, we can see that incorporating preconditioning operator \mathbf{P} incurs a negligible cost to the overall cost of LSRTM. In particular, the preconditioner costs 2 FFTs $O(n\log n)$ while the cost of RTM significantly surpasses that. See the Appendix for the complete code of preconditioner (Mtf2dprec.c).

```

22 def conjgrad(oper,dat,x0,ref,niter):
23     'CG for minimizing |oper x - dat|^2'
24
25     x = x0
26     R = oper(adj=0)[x]-dat
27     for iter in range(niter):
28         g = oper(adj=1)[R] LT
29         g = oper(adj=0)[g] L
30         gn = g.dot2()
31         print "Gradient iter %d: %g" % (iter+1,gn)
32         if 0==iter:
33             s = g
34             S = G
35         else:
36             beta = gn/gnp
37             s = g+s*beta
38             S = G+S*beta
39             gnp = gn
40             alpha = -gn/S.dot2()
41             x = x+s*alpha
42             R = R+S*alpha
43
44
45
46
47     x = prec(adj=0)[p] recover the original variable m = Py
48     return x

```

```

22 def pconjgrad(oper,prec,dat,p0,ref,niter):
23     'Precondition CG for minimizing |oper prec p - dat|'
24
25     p = p0
26     x = prec(adj=0)[p0]
27     R = oper(adj=0)[x]-dat
28     for iter in range(niter):
29         f = oper(adj=1)[R] PTLT
30         g = prec(adj=1)[f] PTL
31         F = prec(adj=0)[g] LP
32         G = oper(adj=0)[f] LP
33         gn = g.dot2()
34         print "Gradient iter %d: %g" % (iter+1,gn)
35         if 0==iter:
36             s = g
37             S = G
38         else:
39             beta = gn/gnp
40             s = g+s*beta
41             S = G+S*beta
42             gnp = gn
43             alpha = -gn/S.dot2()
44             p = p+s*alpha
45             R = R+S*alpha
46
47     x = prec(adj=0)[p] recover the original variable m = Py
48     return x

```

Figure 3.10: Changes in Conjugate-gradient algorithms to incorporate preconditioner
chapter-lsrtm/..../chapter-lsrtm/pre conj

Once we have all the code ready, we perform least-squares reverse time migration using conjugate gradients for 20 iterations. The result without and without preconditioner is shown in Figure 3.11. The LSRTM image with a preconditioner leads to a noticeable improvement in resolution. The amplitudes appear more balanced, and the reflectors at greater depth are better illuminated.

The resolution improvement can be verified by examining the average frequency spectrum of each image (Figure 3.12). LSRTM with preconditioner (pink curve) has the broadest bandwidth compared to LSRTM without preconditioner (red curve) and RTM image (blue curve).

For closer inspection, the selected zoom-in sections are also provided in Figures 3.13,3.14,3.15,3.16

Subsequently, we run LSRTM for 100 iterations. To quantify the improvement, the misfit in data domain (Figure 3.17) is calculated using the L_2 -norm of data residual $\|\mathbf{d}_k - \mathbf{d}\|_2$ normalized by $\|\mathbf{d}\|_2$. Here we also include the result of using only space

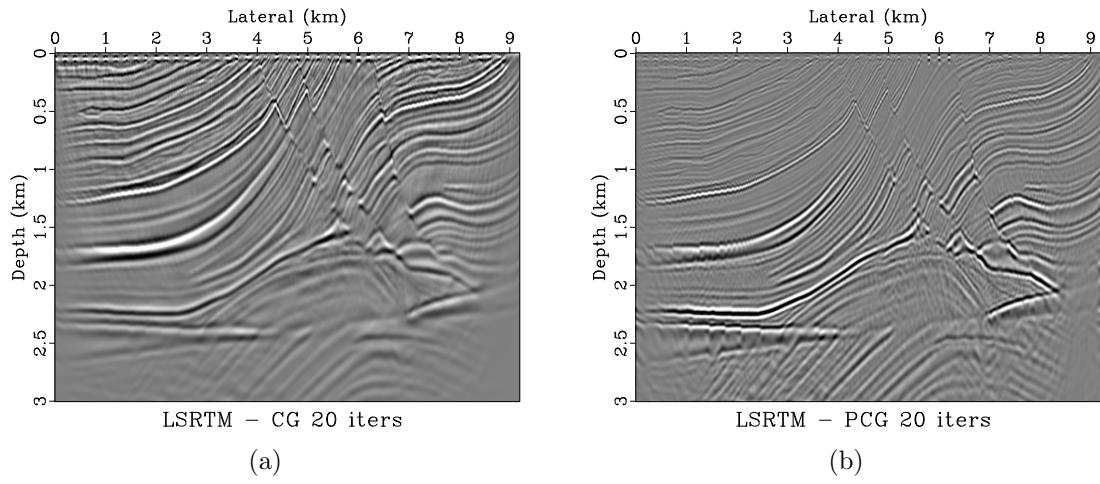


Figure 3.11: Marmousi - (a) LSRTM Image without and (b) with Preconditioner after 20 iterations

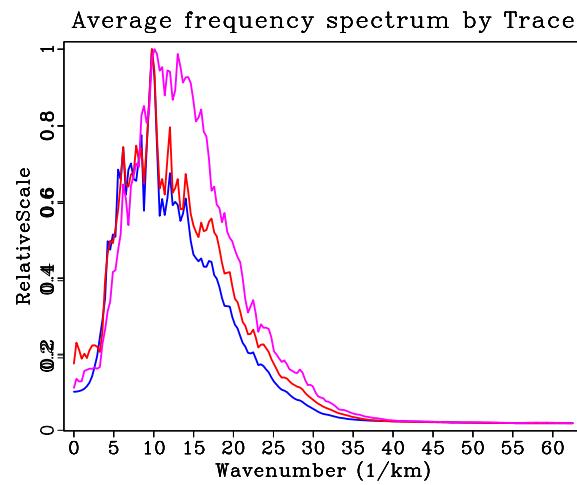


Figure 3.12: Marmousi - Frequency spectrum of migrated image - blue: standard RTM, red: LSRTM, pink: Preconditioned LSRTM

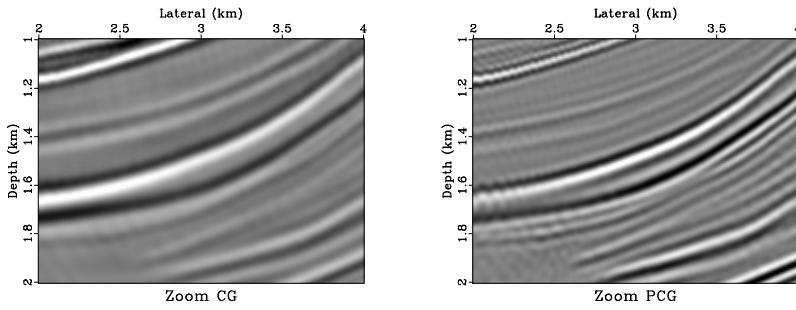


Figure 3.13: Marmousi - Zoom-in comparison 1 between CG and Preconditioned CG
 chapter-lsrtm/.../chapter-lsrtm/pre zm1

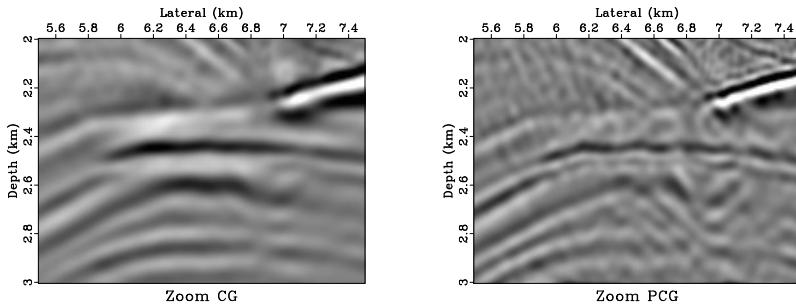


Figure 3.14: Marmousi - Zoom-in comparison 2 comparison between CG and Preconditioned CG
 chapter-lsrtm/.../chapter-lsrtm/pre zm2

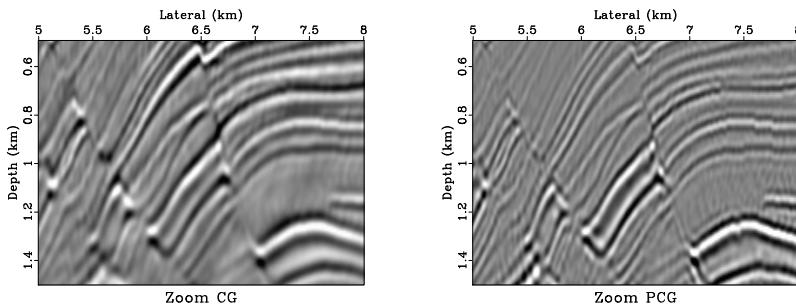


Figure 3.15: Marmousi - Zoom-in comparison 3 comparison between CG and Preconditioned CG
 chapter-lsrtm/.../chapter-lsrtm/pre zm3

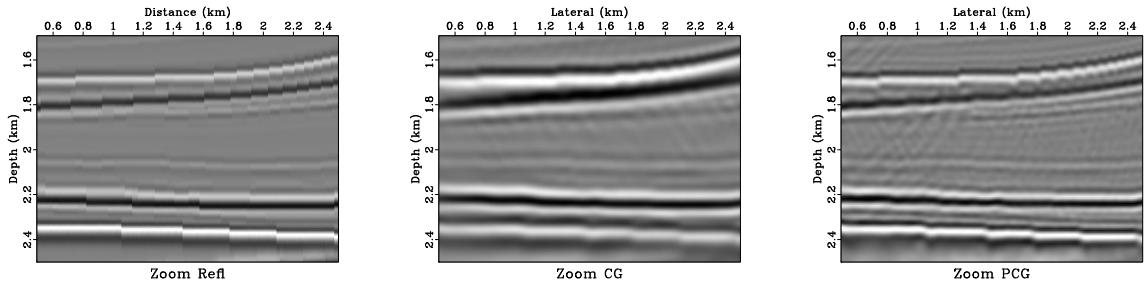


Figure 3.16: Marmousi - Zoom-in comparison 4 comparison between CG and Preconditioned CG

weight \mathbf{W} obtained from taking the square root of the smooth division of $\frac{m_2}{m_1}$ while leaving \mathbf{W}_f as \mathbf{I} as a preconditioner. This is the simplest form of preconditioner one comes up with. The plot confirms that the chain preconditioner leads to faster convergence while using space-only weight shows a slight improvement in convergence. The LSRTM images after 100 iteration are shown in Figures 3.18.

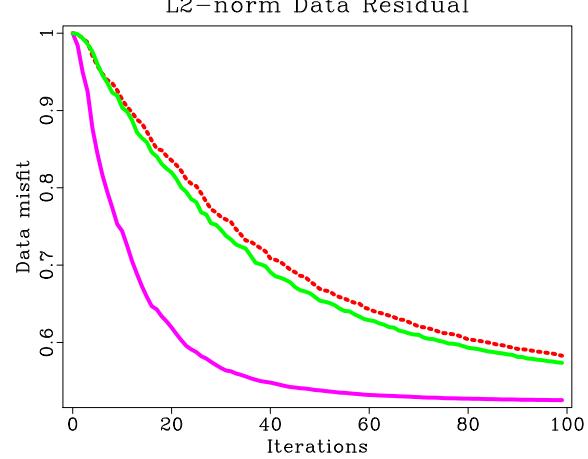


Figure 3.17: Marmousi - Normalized data misfit dash=without preconditioner solid(green)=with weight in space only solid(purple)=with chain preconditioner

In terms of amplitude balancing, LSRTM without preconditioner after 100

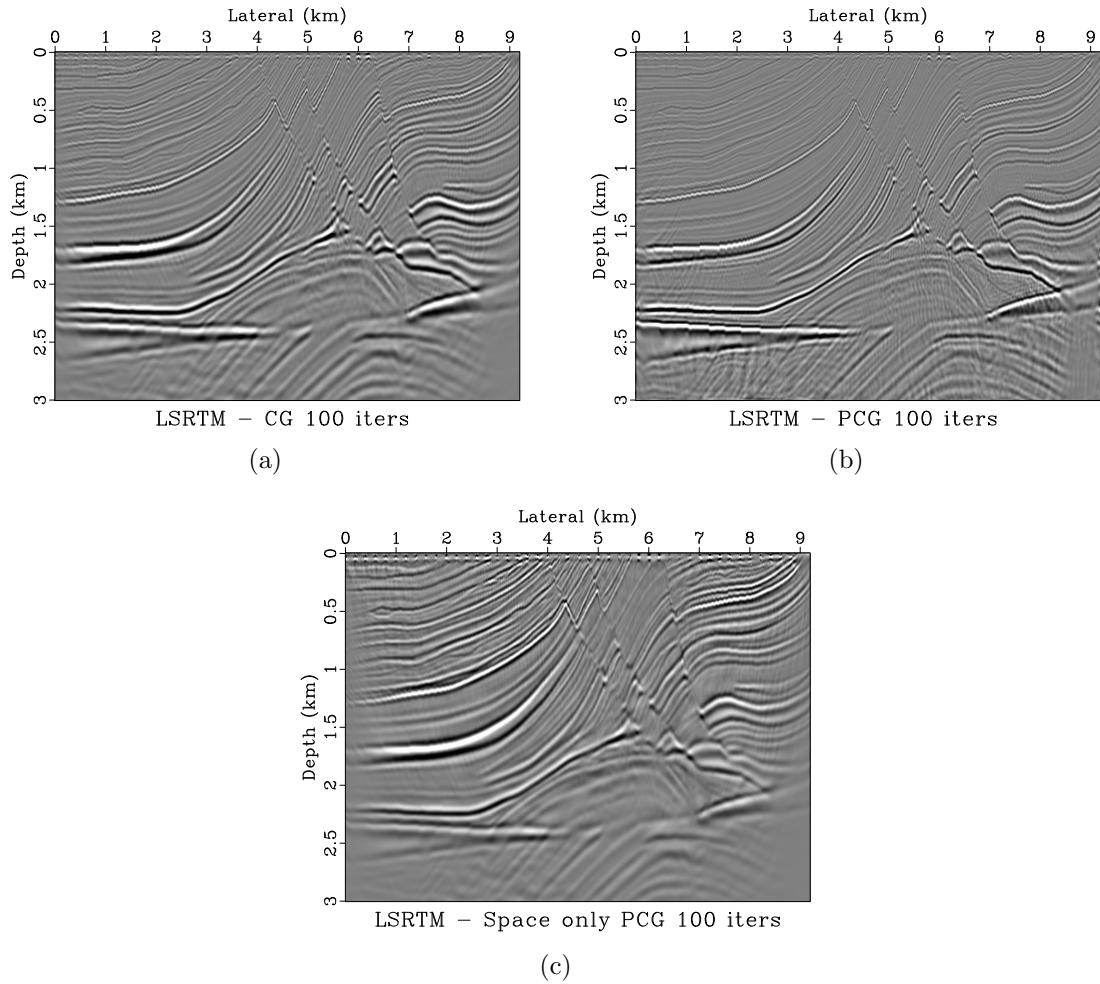


Figure 3.18: Marmousi - (left)LSRTM Image without Preconditioner (mid) with Preconditioner (\mathbf{W}, \mathbf{Wf}) (right) Preconditioner (\mathbf{W} only)after 100 iterations
 chapter-lsrtm/..../chapter-lsrtm/pre cgrad100,pgrad100,wwpgrad100

iterations gives a similar image to the image from LSRTM with preconditioner after only 20 iterations.

Next, we are still testing our method with synthetic data, but now we are using the data with a different geological setting. In the following experiment, we use the Sigsbee2A model which emulates the geological setting of the Gulf of Mexico. It is characterized by the presence of nearly flat layers and a large salt body as shown by the stratigraphic velocity in Fig 3.19. The model was developed by The Subsalt Multiples Attenuation and Reduction Technology Joint Venture(SMAART JV).

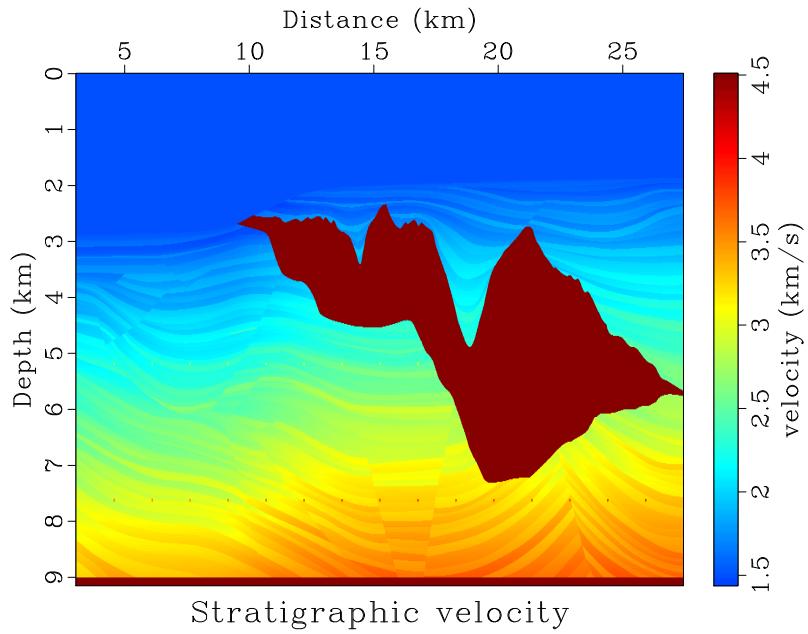


Figure 3.19: Sigsbee - Stratigraphic velocity chapter-lsrtm/..../chapter-lsrtm/sigs veltrue

Starting with the shot record in Fig 3.20, we proceed as before by performing the Reverse-Time Migration. The result before removing low-frequency noise is shown in Fig 3.21(a) while the image after removing it is shown in Fig 3.21(b) . From this initial assessment, we can expect in this dataset, the low frequency predominates the

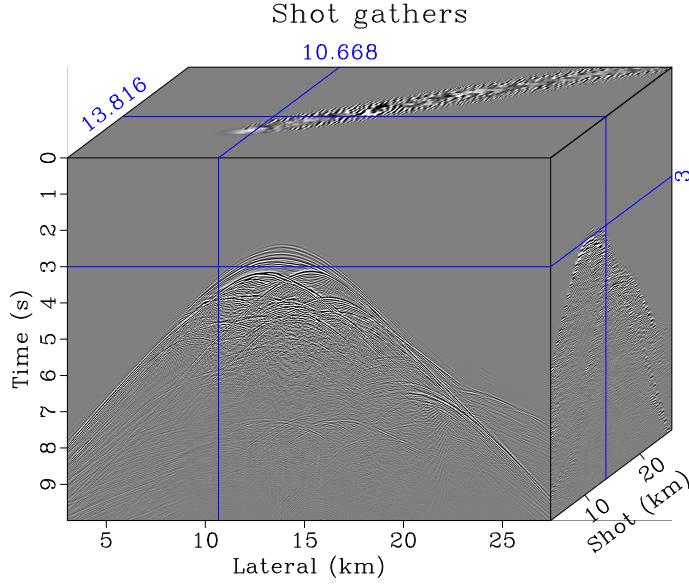


Figure 3.20: Sigsbee - Synthetic shots chapter-lsrtm/..../chapter-lsrtm/sigs bshots45

migrated image and likely to cause difficulties to the chain algorithm in a later step as we are trying to find the frequency weight \mathbf{W}_f .

We solve for weight in space domain \mathbf{W} and frequency domain \mathbf{W}_f shown in Fig 3.22(a), 3.22(b) respectively. We can see that the weight in the frequency domain looks unrealistic because, with the presence of flat layers, the output of the 2D Fourier Transform of the image should appear to be a 90° dip in (k_x, k_z) domain. In fact, this is what we see in 3.4 in case of Marmousi model. We think this is because the artifacts of RTM (low-frequency noise) make it harder for the algorithms to find the true/reasonable weights.

We experiment and perform Least-square RTM. Noted that in the process of LSRTM, we did not apply a low-pass filter to remove low-frequency artifacts in any step of the conjugate gradient. The result of LSRTM without preconditioner and with preconditioner is shown in Fig 3.23 and 3.24 respectively.

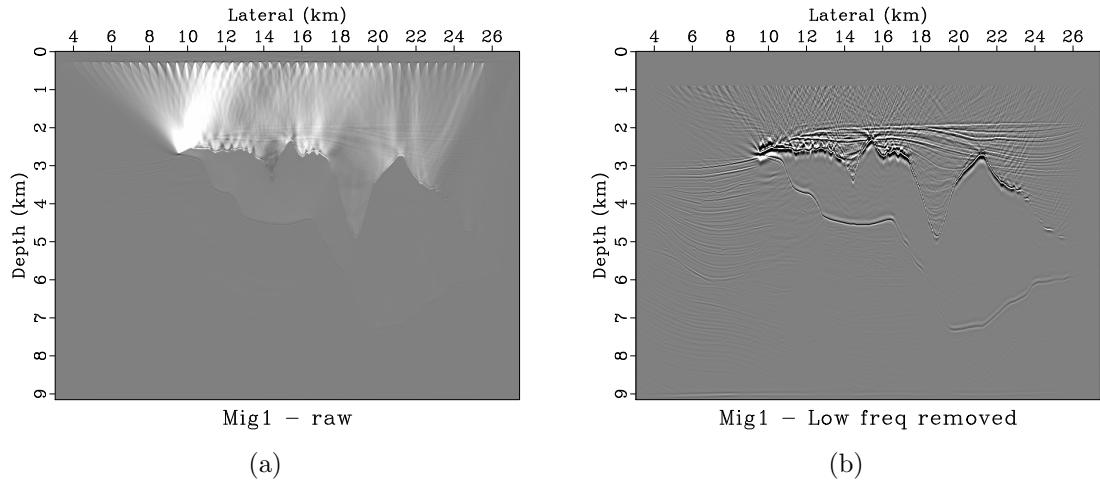


Figure 3.21: Sigsbee - RTM image (a) before remove low-frequency noise (b) after remove low-frequency noise

[chapter-lsrtm/..//chapter-lsrtm/sigs smig1,bmig1](#)

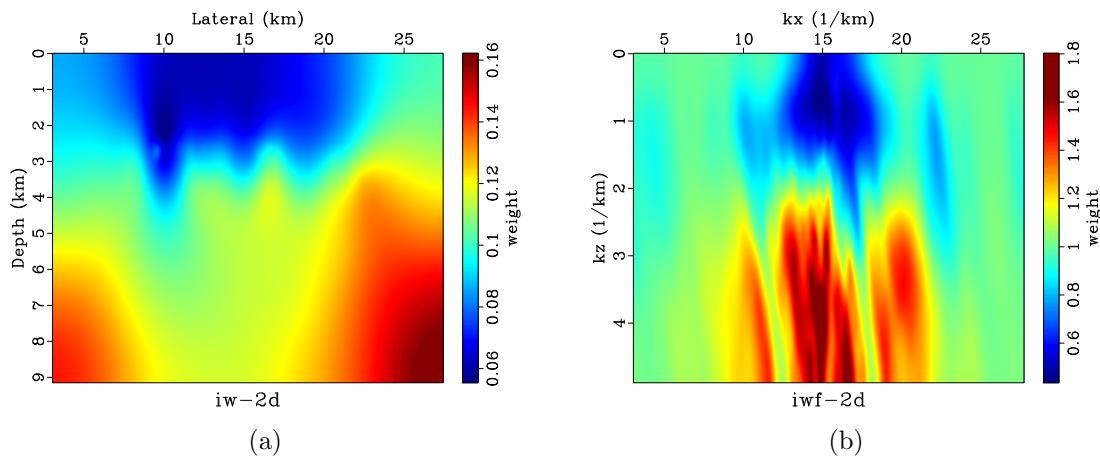


Figure 3.22: Sigsbee - (a) Weight in space domain \mathbf{W} (b) in Frequency domain \mathbf{W}_f

[chapter-lsrtm/..//chapter-lsrtm/sigs iw-2d,iwf-2d](#)

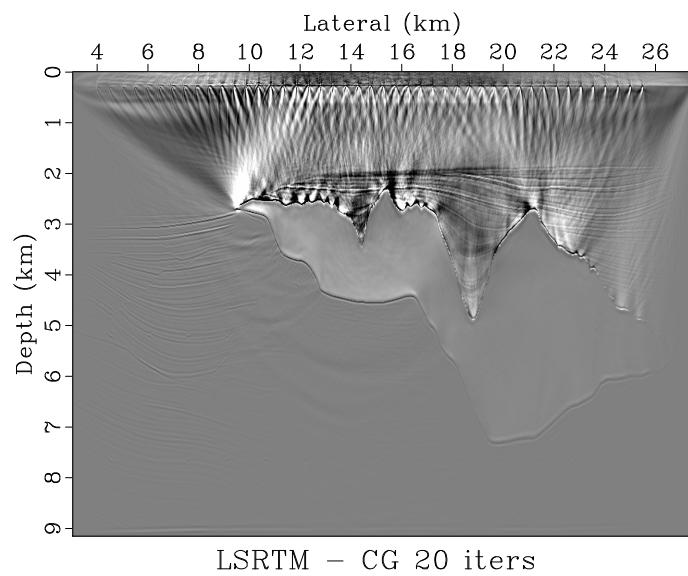


Figure 3.23: Sigsbee - LSRTM image without preconditioner
chapter-lsrtm/..../chapter-lsrtm/sigs cgrad-out0

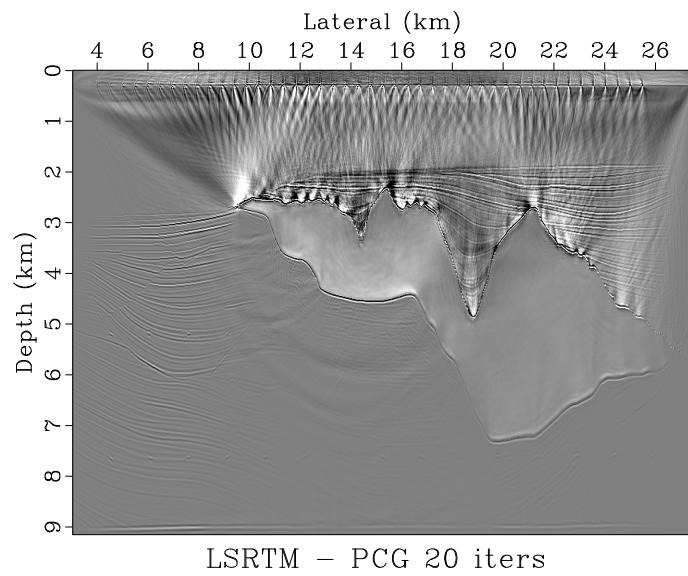


Figure 3.24: Sigsbee - LSRTM image with preconditioner
chapter-lsrtm/..../chapter-lsrtm/sigs pcgrad-out0

From these Figures 3.23 and 3.24, we can see that Least-squares RTM has trouble getting attenuating the low-frequency noise in the area above the salt body. However, to the left of the image where there is no salt body, LSRTM with preconditioner gives better illumination. In fact, below the salt body around 12-16 km, preconditioned LSRTM also gives slightly better illumination.

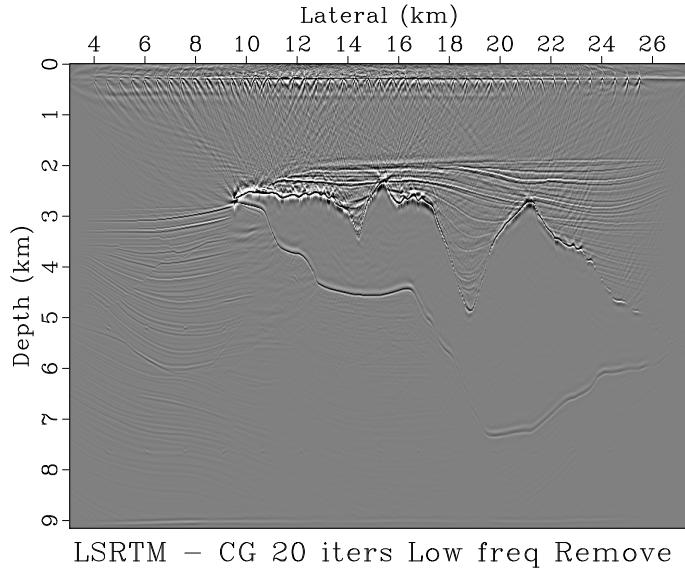


Figure 3.25: Sigsbee - LSRTM image without preconditioner after remove low frequency
 chapter-lsrtm/..//chapter-lsrtm/sigs_lsrtm0

Fig 3.25 and 3.26 show the LSRTM after remove the low frequency noise. Here, we see that the preconditioner has a very small positive effect on the final image, far less effective than the case of the Marmousi data set. As mentioned above, we attribute ineffectiveness to the difficulty in dealing with low-frequency noise originated from the nature of RTM algorithms itself along with the presence of a salt body. Given Sigsbee's complex geological setting, we do not practically have any alternatives to RTM as we must rely on two-way wavefield migration to correctly and sufficiently image geological features.

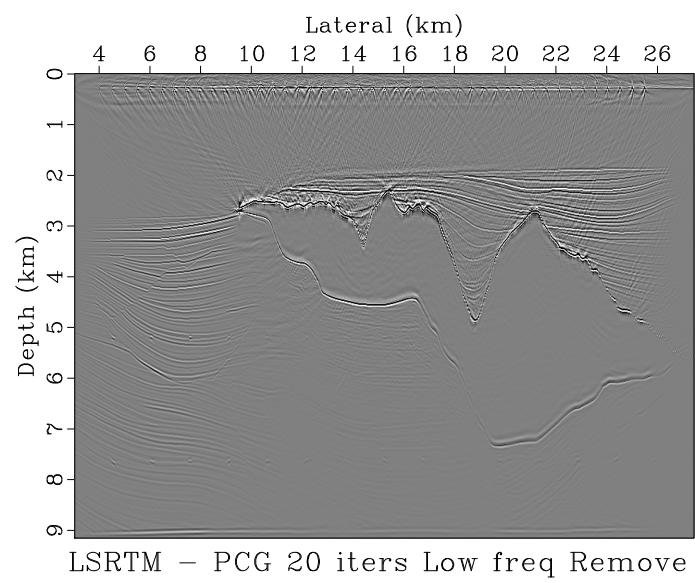


Figure 3.26: Sigsbee - LSRTM image with preconditioner after remove low frequency
chapter-lsrtm/..../chapter-lsrtm/sigs plsrtm0

Chapter 4

Conclusions

In this work, I have proposed a novel approach to approximating the inverse Hessian operator in least-squares migration. The method is data-driven and involves solving for weights in the original and Fourier domains that match two subsequent migrated images using the proposed method "Chain of Operators". The approximated inverse Hessian can be applied either as a non-stationary deconvolution filter to the migrated image or used to form a preconditioner for iterative prestack least-squares migration.

My synthetic-data experiments show promising results in some cases and also reveals the challenges to the usability and limits of the algorithm. In the case that the algorithm works well, the results in the final image have significantly improved resolution, balanced amplitudes, and a better data fit.

Overall, this work serves as a proof of concept of our proposed method Chain of Operators. It shows that the complex operator can be effectively approximated through the chain of parameterized elementary operators - at least in the application of seismic least-square migration. The same principle works in machine learning with artificial neural networks, at a higher cost.

FUTURE WORK

For this particular application, we hope to further investigate the effectiveness of the algorithm with the real field-size prestack datasets. It is also worth studying alternative schemes of solving for the chain weights beside the regularized Gauss-Newton approach.

In the field of seismic processing, the idea of the chain of operators may find many other promising applications. One area is to find the seismic shift between two seismic datasets (Liner and Clapp, 2004). We can represent a small shift in the seismic trace by one elementary operator. Then, the bigger shift will become the chain of these small shift operators.

It is also worth emphasizing that the idea of the chain of operators is not limited to the field of seismic processing. As we can see from the motivation, it follows from the generality of the hypothesis that "the complex operator can be effectively approximated through the chain of parameterized elementary operators".

Appendix

CODE

Below is the source of the code of the programs that were newly developed for this project. In particular, Mchain2dfft.c and chain2dfft.c are the programs for solving for weights in space and frequency domain. Mtf2dprec.c and tf2dprec.c are the subroutine that is used to implement preconditioning in a Conjugate-gradient solver. The library used in those can be found at Madagascar open-source software environment for reproducible computational experiments (Fomel et al., 2013). The package is available at <http://www.ahay.org/>.

Listing 1: chapter-lsrtm/code/Mchain2dfft.c

```
1  /* Find a symmetric chain of 2D-Fourier weighting and scaling with movies*/
2  /*
3   Copyright (C) 2004 University of Texas at Austin
4
5   This program is free software; you can redistribute it and/or modify
6   it under the terms of the GNU General Public License as published by
7   the Free Software Foundation; either version 2 of the License, or
8   (at your option) any later version.
9
10  This program is distributed in the hope that it will be useful,
11  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  GNU General Public License for more details.
14
15  You should have received a copy of the GNU General Public License
16  along with this program; if not, write to the Free Software
17  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
18 */
19
20 #include <rsf.h>
21 #include "chain2dfft.h"
22 #include "twosmooth2.h"
23 #include "fft2.h"
24
25
26
27 int main(int argc, char* argv[])
28 {
29     int i, n, nk, n2, iter, niter, liter, nt, nx, nt1, nt2, nx2;
30     int rect1, rect2, frect1, frect2;
31     float dt, dx, x0;
32     float l2_r, l2_r_new, alpha;
33     float *w, *dw, *x, *y, *r, *p, *r_new, *w_prev;
34     sf_file wht, fwht, src, tgt, mch;
35     sf_file w0, wf0;
36     /* For fft2 */
37     bool isCmplx = false;
38     int pad = 1;
39 }
```

```

41     sf_init(argc,argv);
42     src = sf_input("in");
43     wht = sf_output("out");
44     /* space weight */

45     tgt = sf_input("target");
46     /* target */

47     w0 = sf_input("init_w");
48     wf0 = sf_input("init_wf");

49     fwht = sf_output("fweight");
50     /* frequency weight */
51     mch = sf_output("match");
52     /* matched */

53

54

55

56     if (!sf_histint(src,"n1",&nt)) sf_error("No n1= in input");
57     if (!sf_histint(src,"n2",&nx)) sf_error("No n2= in input");
58     if (!sf_histfloat(src,"d1",&dt)) sf_error("No d1= in input");
59     if (!sf_histfloat(src,"d2",&dx)) sf_error("No d2= in input");
60     if (!sf_histfloat(src,"o2",&x0)) x0=0.;

61

62     if (!sf_getint("niter",&niter)) niter=0;
63     /* number of iterations */
64     if (!sf_getint("liter",&liter)) liter=50;
65     /* number of linear iterations */

66

67

68     n = nt*nx;

69

70     nk = fft2_init(isCmplx, pad, nt, nx, &nt1, &nx2);
71     nt2 = nk/nx2;

72

73     n2 = 3*n+nk;
74     sf_putint(fwht,"n1",nt2);
75     sf_putint(fwht,"n2",nx2);

76

77     w = sf_floatalloc(n2);
78     dw = sf_floatalloc(n2);
79     w_prev = sf_floatalloc(n2);

80

81     x = sf_floatalloc(n);
82     y = sf_floatalloc(n);
83     r = sf_floatalloc(3*n);
84     r_new = sf_floatalloc(3*n);

85

86

87     if (!sf_getint("rect1",&rect1)) rect1=1;
88     /* smoothing in time dim1*/
89     if (!sf_getint("rect2",&rect2)) rect2=1;
90     /* smoothing in time dim2*/
91     if (!sf_getint("frect1",&frect1)) frect1=1;
92     /* smoothing in frequency dim1 */
93     if (!sf_getint("frect2",&frect2)) frect2=1;
94     /* smoothing in frequency dim2 */

95

96     twosmooth2_init(n,nk,nt,nt2,
97                      rect1,rect2,
98                      frect1,frect2,
99                      2*n);

```

```

101    sf_floatread(x,n,src); /* source */
103    sf_floatread(y,n,tgt); /* target */

105    sfchain2d_init(nt,nx,nt1,nx2,nk,
106                    w+2*n,w+3*n,w,w+n,x);
107
109    sf_conjgrad_init(n2, n2, 3*n, 3*n, 1., 1.e-6, true, false);

111    p = sf_floatalloc(n2);

113    /* initialize w [time w and freqz w] */
114    for (i=0; i < 2*n; i++) {
115        w[i] = 0.0f;
116    }

117    sf_floatread(w+2*n, n, w0);
118    sf_floatread(w+3*n, nk, wf0);

119    for (iter=0; iter < niter; iter++) {
120        sf_warning("Start %d", iter);

121        for (i=0; i < n2; i++) {
122            w_prev[i] = w[i];
123        }
124        alpha=1.0;

125        sfchain2d_res(y,r);
126        l2_r = cblas_snrm2(3*n,r,1);
127        sf_warning("(Before update) L2 norm of res: %g", l2_r);

128        sf_warning("Residual %d", iter);

129        sf_conjgrad(NULL, sfchain2d_lop,twosmooth2_lop,p,dw,r,liter);

130        for (i=0; i < n2; i++) {
131            w[i] += alpha*dw[i];
132        }

133        sfchain2d_res(y,r_new);

134        l2_r_new = cblas_snrm2(3*n,r_new,1);

135        sf_warning("(After update) L2 norm of res: %g, alpha = %g", l2_r_new,alpha);

136        while(l2_r_new>l2_r && alpha > 0.00001){
137            sf_warning("Too big step !");
138
139            for (i=0; i < n2; i++) {
140                w[i] = w_prev[i];
141            }

142            alpha *=0.5;

143            for (i=0; i < n2; i++) {
144                w[i] += alpha*dw[i];
145            }
146        }
147    }
148
149    sf_floatwrite(y,n,tgt);
150
151    sf_close(src);
152    sf_close(tgt);
153
154    return 0;
155
156    /* clean up */
157
158    sf_floatfree(p);
159
160    return 0;
161

```

```

163     sfchain2d_res(y,r_new);
164
165     l2_r_new = cblas_snrm2(3*n,r_new,1);
166     sf_warning("(After update) L2 norm of res: %g, alpha = %g", l2_r_new,alpha);
167
168 }
169 sf_warning("Pass now !");
170
171
172
173
174
175 /* End of iteration */
176
177 sf_floatwrite(w+2*n,n,wht);
178 sf_floatwrite(w+3*n,nk, fwht);
179 sfchain2d_apply(y);
180 sf_floatwrite(y,n,mch);
181
182 exit(0);
183 }
```

Listing 2: chapter-lsrtm/code/chain2dfft.c

```

37         float *y2    /* intermediate [n1*n2] */,
38         float *src   /* source [n1*n2] */
39 /*< initialize >*
40 {
41     nt = n1;
42     nx = n2;
43     nt1 = n1pad;
44     nx2 = n_fftx;
45     nk = n_out_fft;
46     nt2 = nk/n_fftx;
47     nw = nt2;
48
49     /*Model parameters*/
50     n = n1*n2;
51     w = w1;
52     wf = wf1;
53     x1 = y1;
54     x2 = y2;
55     s = src;
56
56     tmp1 = sf_floatalloc2(nt1,nx2);
57     tmp2 = sf_floatalloc2(nt1,nx2);
58
59     /*for 2D fft*/
60     ctmp1 = sf_complexalloc(nk);
61     ctmp2 = sf_complexalloc(nk);
62
63 }
64
65 void sfchain2d_close(void)
66 /*< clean allocated storage >*
67 {
68     free(*tmp1);
69     free(*tmp2);
70     free(tmp1);
71     free(tmp2);
72     free(ctmp1);
73     free(ctmp2);
74 }
75
76 void sfchain2d_res(const float *t /* target */,
77                     float *r          /* residual */)
78 /*< apply the chain operator >*
79 {
80     int i, i1, i2, ik;
81
82     /* pad with zeros */
83     for (i2=0; i2 < nx; i2++) {
84         for (i1=0; i1 < nt; i1++) {
85             tmp2[i2][i1] = x1[i1+i2*nt];
86         }
87         for (i1=nt; i1 < nt1; i1++) {
88             tmp2[i2][i1] = 0.0f;
89         }
90     }
91     for (i2=nx; i2 < nx2; i2++) {
92         for (i1=0; i1 < nt1; i1++) {
93             tmp2[i2][i1] = 0.0f;
94         }
95     }
96 }
97

```

```

99  /* forward FFT */
101 fft2_allocate(ctmp2);
102 fft2(tmp2[0], ctmp2);
103
104 /* frequency weight */
105 for (ik=0; ik < nk; ik++) {
106 ctmp2[ik] *= wf[ik];
107 }
108 /* inverse FFT */
109 ifft2(tmp2[0], ctmp2);
110 /* Compute residual r */
111 for(i=0; i<nt*nx; i++){
112 i1 = i%nt;
113 i2 = i/nt;
114
115 r[i] = x1[i]-w[i]*s[i];
116 r[n+i] = x2[i]-tmp2[i2][i1];
117 r[2*n+i] = t[i]-w[i]*x2[i];
118 }
119 }
120
121 void sfchain2d_apply(float *y)
122 /*< apply the chain operator >*/
123 {
124 int i, ik, i1, i2;
125
126 /* pad with zeros */
127 for (i2=0; i2 < nx; i2++) {
128 for (i1=0; i1 < nt; i1++) {
129 i = i1+i2*nt;
130 tmp2[i2][i1] = w[i]*s[i];
131 }
132 for (i1=nt; i1 < nt1; i1++) {
133 tmp2[i2][i1] = 0.0f;
134 }
135 }
136 for (i2=nx; i2 < nx2; i2++) {
137 for (i1=0; i1 < nt1; i1++) {
138 tmp2[i2][i1] = 0.0f;
139 }
140 }
141
142 /* forward FFT */
143 fft2_allocate(ctmp2);
144 fft2(tmp2[0], ctmp2);
145
146 /* frequency weight */
147 for (ik=0; ik < nk; ik++) {
148 ctmp2[ik] *= wf[ik];
149 }
150 /* inverse FFT */
151 ifft2(tmp2[0], ctmp2);
152
153 for (i=0; i < n; i++) {
154 i1 = i%nt;
155 i2 = i/nt;
156
157 y[i] = w[i]*tmp2[i2][i1];

```

```

159     }
160 }
161
163
165 void sfchain2d_lop (bool adj, bool add, int nxx, int nyy, float* x, float* y)
166 /*< linear operator >*/
167 {
168     int i, ik, i1, i2;
169     int *dc_id, *nyq_id;
170     int id;
171     float scale;
172     dc_id = sf_intalloc(nx);
173     nyq_id = sf_intalloc(nx);
174
175
176     if (nxx != 3*n+nk || nyy != 3*n) sf_error("%s: Wrong size", __FILE__);
177
178     sf_adjnull(adj, add, nxx, nyy, x, y);
179
180     if (adj) {
181         for (i=0; i < n; i++) {
182             i1 = i%nt;
183             i2 = i/nt;
184
185             tmp1[i2][i1] = y[n+i];
186             tmp2[i2][i1] = x1[i];
187
188             x[n+i] += w[i]*y[2*n+i] - y[n+i];
189             x[2*n+i] += x2[i]*y[2*n+i];
190         }
191
192         /* pad with zeros */
193         for (i2=0; i2 < nx; i2++) {
194             for (i1=nt; i1 < nt1; i1++) {
195                 tmp1[i2][i1] = 0.0f;
196                 tmp2[i2][i1] = 0.0f;
197             }
198         }
199         for (i2=nx; i2 < nx2; i2++) {
200             for (i1=0; i1 < nt1; i1++) {
201                 tmp1[i2][i1] = 0.0f;
202                 tmp2[i2][i1] = 0.0f;
203             }
204         }
205
206         fft2_allocate(ctmp2);
207         fft2(tmp2[0], ctmp2);
208         fft2_allocate(ctmp1);
209         fft2(tmp1[0], ctmp1);
210
211         for(id = 0; id < nx; id++){
212             dc_id[id] = id*nw;
213             nyq_id[id] = (id+1)*nw - 1;
214         }
215
216         for (ik=0; ik < nk; ik++) {
217             scale = 2.0;
218             for(id=0; id<nx; id++)
219

```

```

221
222     {
223         if(ik==dc_id[id] || ik == nyq_id[id]){
224             scale = 1.0;
225             break;
226         }
227     }
228
229     x[3*n+ik] += scale*crealf(ctmp1[ik]*conjf(ctmp2[ik])/(nt1*nx2));
230
231     ctmp1[ik] *= wf[ik];
232 }
233 ifft2(tmp1[0],ctmp1);
234
235 for (i=0; i < n; i++) {
236     i1 = i%nt;
237     i2 = i/nt;
238
239     x[2*n+i] += s[i]*y[i];
240     x[i] += tmp1[i2][i1] - y[i];
241 }
242 } /*Forward*/
243
244 for (i=0; i < n; i++) {
245     i1 = i%nt;
246     i2 = i/nt;
247
248     y[i] += s[i]*x[2*n+i] - x[i];
249     tmp1[i2][i1] = x[i];
250     tmp2[i2][i1] = x1[i];
251 }
252 /* pad with zeros */
253 for (i2=0; i2 < nx; i2++) {
254     for (i1=nt; i1 < nt1; i1++) {
255         tmp1[i2][i1] = 0.0f;
256         tmp2[i2][i1] = 0.0f;
257     }
258 }
259 for (i2=nx; i2 < nx2; i2++) {
260     for (i1=0; i1 < nt1; i1++) {
261         tmp1[i2][i1] = 0.0f;
262         tmp2[i2][i1] = 0.0f;
263     }
264 }
265
266 fft2_allocate(ctmp1);
267 fft2(tmp1[0],ctmp1);
268 for(ik=0; ik<nk;ik++){
269     ctmp1[ik] *=wf[ik];
270 }
271 ifft2(tmp1[0],ctmp1);
272 fft2_allocate(ctmp2);
273 fft2(tmp2[0],ctmp2);
274 for(ik=0; ik<nk;ik++){
275     ctmp2[ik] *=x[3*n+ik];
276 }
277 ifft2(tmp2[0],ctmp2);
278
279 for (i=0; i < n; i++) {
280     i1 = i%nt;
281     i2 = i/nt;
282
283     x[3*n+i] += scale*crealf(ctmp1[ik]*conjf(ctmp2[ik])/(nt1*nx2));
284     ctmp1[ik] *= wf[ik];
285 }
286 }
287
288 for (i=0; i < n; i++) {
289     i1 = i%nt;
290     i2 = i/nt;
291
292     y[i] += s[i]*x[2*n+i] - x[i];
293     tmp1[i2][i1] = x[i];
294     tmp2[i2][i1] = x1[i];
295 }
296
297 /* pad with zeros */
298 for (i2=0; i2 < nx; i2++) {
299     for (i1=nt; i1 < nt1; i1++) {
300         tmp1[i2][i1] = 0.0f;
301         tmp2[i2][i1] = 0.0f;
302     }
303 }
304 for (i2=nx; i2 < nx2; i2++) {
305     for (i1=0; i1 < nt1; i1++) {
306         tmp1[i2][i1] = 0.0f;
307         tmp2[i2][i1] = 0.0f;
308     }
309 }
310 }
311
312 fft2_allocate(ctmp1);
313 fft2(tmp1[0],ctmp1);
314 for(ik=0; ik<nk;ik++){
315     ctmp1[ik] *=wf[ik];
316 }
317 ifft2(tmp1[0],ctmp1);
318 fft2_allocate(ctmp2);
319 fft2(tmp2[0],ctmp2);
320 for(ik=0; ik<nk;ik++){
321     ctmp2[ik] *=x[3*n+ik];
322 }
323 ifft2(tmp2[0],ctmp2);
324
325 for (i=0; i < n; i++) {
326     i1 = i%nt;
327     i2 = i/nt;
328
329     y[i] += s[i]*x[2*n+i] - x[i];
330     tmp1[i2][i1] = x[i];
331     tmp2[i2][i1] = x1[i];
332 }
333
334 /* pad with zeros */
335 for (i2=0; i2 < nx; i2++) {
336     for (i1=nt; i1 < nt1; i1++) {
337         tmp1[i2][i1] = 0.0f;
338         tmp2[i2][i1] = 0.0f;
339     }
340 }
341 for (i2=nx; i2 < nx2; i2++) {
342     for (i1=0; i1 < nt1; i1++) {
343         tmp1[i2][i1] = 0.0f;
344         tmp2[i2][i1] = 0.0f;
345     }
346 }
347 }
348
349 fft2_allocate(ctmp1);
350 fft2(tmp1[0],ctmp1);
351 for(ik=0; ik<nk;ik++){
352     ctmp1[ik] *=wf[ik];
353 }
354 ifft2(tmp1[0],ctmp1);
355 fft2_allocate(ctmp2);
356 fft2(tmp2[0],ctmp2);
357 for(ik=0; ik<nk;ik++){
358     ctmp2[ik] *=x[3*n+ik];
359 }
360 ifft2(tmp2[0],ctmp2);
361
362 for (i=0; i < n; i++) {
363     i1 = i%nt;
364     i2 = i/nt;
365
366     y[i] += s[i]*x[2*n+i] - x[i];
367     tmp1[i2][i1] = x[i];
368     tmp2[i2][i1] = x1[i];
369 }
370
371 /* pad with zeros */
372 for (i2=0; i2 < nx; i2++) {
373     for (i1=nt; i1 < nt1; i1++) {
374         tmp1[i2][i1] = 0.0f;
375         tmp2[i2][i1] = 0.0f;
376     }
377 }
378 for (i2=nx; i2 < nx2; i2++) {
379     for (i1=0; i1 < nt1; i1++) {
380         tmp1[i2][i1] = 0.0f;
381         tmp2[i2][i1] = 0.0f;
382     }
383 }
384 }
385
386 fft2_allocate(ctmp1);
387 fft2(tmp1[0],ctmp1);
388 for(ik=0; ik<nk;ik++){
389     ctmp1[ik] *=wf[ik];
390 }
391 ifft2(tmp1[0],ctmp1);
392 fft2_allocate(ctmp2);
393 fft2(tmp2[0],ctmp2);
394 for(ik=0; ik<nk;ik++){
395     ctmp2[ik] *=x[3*n+ik];
396 }
397 ifft2(tmp2[0],ctmp2);
398
399 for (i=0; i < n; i++) {
400     i1 = i%nt;
401     i2 = i/nt;
402
403     y[i] += s[i]*x[2*n+i] - x[i];
404     tmp1[i2][i1] = x[i];
405     tmp2[i2][i1] = x1[i];
406 }
407
408 /* pad with zeros */
409 for (i2=0; i2 < nx; i2++) {
410     for (i1=nt; i1 < nt1; i1++) {
411         tmp1[i2][i1] = 0.0f;
412         tmp2[i2][i1] = 0.0f;
413     }
414 }
415 for (i2=nx; i2 < nx2; i2++) {
416     for (i1=0; i1 < nt1; i1++) {
417         tmp1[i2][i1] = 0.0f;
418         tmp2[i2][i1] = 0.0f;
419     }
420 }
421 }
422
423 fft2_allocate(ctmp1);
424 fft2(tmp1[0],ctmp1);
425 for(ik=0; ik<nk;ik++){
426     ctmp1[ik] *=wf[ik];
427 }
428 ifft2(tmp1[0],ctmp1);
429 fft2_allocate(ctmp2);
430 fft2(tmp2[0],ctmp2);
431 for(ik=0; ik<nk;ik++){
432     ctmp2[ik] *=x[3*n+ik];
433 }
434 ifft2(tmp2[0],ctmp2);
435
436 for (i=0; i < n; i++) {
437     i1 = i%nt;
438     i2 = i/nt;
439
440     y[i] += s[i]*x[2*n+i] - x[i];
441     tmp1[i2][i1] = x[i];
442     tmp2[i2][i1] = x1[i];
443 }
444
445 /* pad with zeros */
446 for (i2=0; i2 < nx; i2++) {
447     for (i1=nt; i1 < nt1; i1++) {
448         tmp1[i2][i1] = 0.0f;
449         tmp2[i2][i1] = 0.0f;
450     }
451 }
452 for (i2=nx; i2 < nx2; i2++) {
453     for (i1=0; i1 < nt1; i1++) {
454         tmp1[i2][i1] = 0.0f;
455         tmp2[i2][i1] = 0.0f;
456     }
457 }
458 }
459
460 fft2_allocate(ctmp1);
461 fft2(tmp1[0],ctmp1);
462 for(ik=0; ik<nk;ik++){
463     ctmp1[ik] *=wf[ik];
464 }
465 ifft2(tmp1[0],ctmp1);
466 fft2_allocate(ctmp2);
467 fft2(tmp2[0],ctmp2);
468 for(ik=0; ik<nk;ik++){
469     ctmp2[ik] *=x[3*n+ik];
470 }
471 ifft2(tmp2[0],ctmp2);
472
473 for (i=0; i < n; i++) {
474     i1 = i%nt;
475     i2 = i/nt;
476
477     y[i] += s[i]*x[2*n+i] - x[i];
478     tmp1[i2][i1] = x[i];
479     tmp2[i2][i1] = x1[i];
480 }
481
482 /* pad with zeros */
483 for (i2=0; i2 < nx; i2++) {
484     for (i1=nt; i1 < nt1; i1++) {
485         tmp1[i2][i1] = 0.0f;
486         tmp2[i2][i1] = 0.0f;
487     }
488 }
489 for (i2=nx; i2 < nx2; i2++) {
490     for (i1=0; i1 < nt1; i1++) {
491         tmp1[i2][i1] = 0.0f;
492         tmp2[i2][i1] = 0.0f;
493     }
494 }
495 }
496
497 fft2_allocate(ctmp1);
498 fft2(tmp1[0],ctmp1);
499 for(ik=0; ik<nk;ik++){
500     ctmp1[ik] *=wf[ik];
501 }
502 ifft2(tmp1[0],ctmp1);
503 fft2_allocate(ctmp2);
504 fft2(tmp2[0],ctmp2);
505 for(ik=0; ik<nk;ik++){
506     ctmp2[ik] *=x[3*n+ik];
507 }
508 ifft2(tmp2[0],ctmp2);
509
510 for (i=0; i < n; i++) {
511     i1 = i%nt;
512     i2 = i/nt;
513
514     y[i] += s[i]*x[2*n+i] - x[i];
515     tmp1[i2][i1] = x[i];
516     tmp2[i2][i1] = x1[i];
517 }
518
519 /* pad with zeros */
520 for (i2=0; i2 < nx; i2++) {
521     for (i1=nt; i1 < nt1; i1++) {
522         tmp1[i2][i1] = 0.0f;
523         tmp2[i2][i1] = 0.0f;
524     }
525 }
526 for (i2=nx; i2 < nx2; i2++) {
527     for (i1=0; i1 < nt1; i1++) {
528         tmp1[i2][i1] = 0.0f;
529         tmp2[i2][i1] = 0.0f;
530     }
531 }
532 }
533
534 fft2_allocate(ctmp1);
535 fft2(tmp1[0],ctmp1);
536 for(ik=0; ik<nk;ik++){
537     ctmp1[ik] *=wf[ik];
538 }
539 ifft2(tmp1[0],ctmp1);
540 fft2_allocate(ctmp2);
541 fft2(tmp2[0],ctmp2);
542 for(ik=0; ik<nk;ik++){
543     ctmp2[ik] *=x[3*n+ik];
544 }
545 ifft2(tmp2[0],ctmp2);
546
547 for (i=0; i < n; i++) {
548     i1 = i%nt;
549     i2 = i/nt;
550
551     y[i] += s[i]*x[2*n+i] - x[i];
552     tmp1[i2][i1] = x[i];
553     tmp2[i2][i1] = x1[i];
554 }
555
556 /* pad with zeros */
557 for (i2=0; i2 < nx; i2++) {
558     for (i1=nt; i1 < nt1; i1++) {
559         tmp1[i2][i1] = 0.0f;
560         tmp2[i2][i1] = 0.0f;
561     }
562 }
563 for (i2=nx; i2 < nx2; i2++) {
564     for (i1=0; i1 < nt1; i1++) {
565         tmp1[i2][i1] = 0.0f;
566         tmp2[i2][i1] = 0.0f;
567     }
568 }
569 }
570
571 fft2_allocate(ctmp1);
572 fft2(tmp1[0],ctmp1);
573 for(ik=0; ik<nk;ik++){
574     ctmp1[ik] *=wf[ik];
575 }
576 ifft2(tmp1[0],ctmp1);
577 fft2_allocate(ctmp2);
578 fft2(tmp2[0],ctmp2);
579 for(ik=0; ik<nk;ik++){
580     ctmp2[ik] *=x[3*n+ik];
581 }
582 ifft2(tmp2[0],ctmp2);
583
584 for (i=0; i < n; i++) {
585     i1 = i%nt;
586     i2 = i/nt;
587
588     y[i] += s[i]*x[2*n+i] - x[i];
589     tmp1[i2][i1] = x[i];
590     tmp2[i2][i1] = x1[i];
591 }
592
593 /* pad with zeros */
594 for (i2=0; i2 < nx; i2++) {
595     for (i1=nt; i1 < nt1; i1++) {
596         tmp1[i2][i1] = 0.0f;
597         tmp2[i2][i1] = 0.0f;
598     }
599 }
600 for (i2=nx; i2 < nx2; i2++) {
601     for (i1=0; i1 < nt1; i1++) {
602         tmp1[i2][i1] = 0.0f;
603         tmp2[i2][i1] = 0.0f;
604     }
605 }
606 }
607
608 fft2_allocate(ctmp1);
609 fft2(tmp1[0],ctmp1);
610 for(ik=0; ik<nk;ik++){
611     ctmp1[ik] *=wf[ik];
612 }
613 ifft2(tmp1[0],ctmp1);
614 fft2_allocate(ctmp2);
615 fft2(tmp2[0],ctmp2);
616 for(ik=0; ik<nk;ik++){
617     ctmp2[ik] *=x[3*n+ik];
618 }
619 ifft2(tmp2[0],ctmp2);
620
621 for (i=0; i < n; i++) {
622     i1 = i%nt;
623     i2 = i/nt;
624
625     y[i] += s[i]*x[2*n+i] - x[i];
626     tmp1[i2][i1] = x[i];
627     tmp2[i2][i1] = x1[i];
628 }
629
630 /* pad with zeros */
631 for (i2=0; i2 < nx; i2++) {
632     for (i1=nt; i1 < nt1; i1++) {
633         tmp1[i2][i1] = 0.0f;
634         tmp2[i2][i1] = 0.0f;
635     }
636 }
637 for (i2=nx; i2 < nx2; i2++) {
638     for (i1=0; i1 < nt1; i1++) {
639         tmp1[i2][i1] = 0.0f;
640         tmp2[i2][i1] = 0.0f;
641     }
642 }
643 }
644
645 fft2_allocate(ctmp1);
646 fft2(tmp1[0],ctmp1);
647 for(ik=0; ik<nk;ik++){
648     ctmp1[ik] *=wf[ik];
649 }
650 ifft2(tmp1[0],ctmp1);
651 fft2_allocate(ctmp2);
652 fft2(tmp2[0],ctmp2);
653 for(ik=0; ik<nk;ik++){
654     ctmp2[ik] *=x[3*n+ik];
655 }
656 ifft2(tmp2[0],ctmp2);
657
658 for (i=0; i < n; i++) {
659     i1 = i%nt;
660     i2 = i/nt;
661
662     y[i] += s[i]*x[2*n+i] - x[i];
663     tmp1[i2][i1] = x[i];
664     tmp2[i2][i1] = x1[i];
665 }
666
667 /* pad with zeros */
668 for (i2=0; i2 < nx; i2++) {
669     for (i1=nt; i1 < nt1; i1++) {
670         tmp1[i2][i1] = 0.0f;
671         tmp2[i2][i1] = 0.0f;
672     }
673 }
674 for (i2=nx; i2 < nx2; i2++) {
675     for (i1=0; i1 < nt1; i1++) {
676         tmp1[i2][i1] = 0.0f;
677         tmp2[i2][i1] = 0.0f;
678     }
679 }
680 }
681
682 fft2_allocate(ctmp1);
683 fft2(tmp1[0],ctmp1);
684 for(ik=0; ik<nk;ik++){
685     ctmp1[ik] *=wf[ik];
686 }
687 ifft2(tmp1[0],ctmp1);
688 fft2_allocate(ctmp2);
689 fft2(tmp2[0],ctmp2);
690 for(ik=0; ik<nk;ik++){
691     ctmp2[ik] *=x[3*n+ik];
692 }
693 ifft2(tmp2[0],ctmp2);
694
695 for (i=0; i < n; i++) {
696     i1 = i%nt;
697     i2 = i/nt;
698
699     y[i] += s[i]*x[2*n+i] - x[i];
700     tmp1[i2][i1] = x[i];
701     tmp2[i2][i1] = x1[i];
702 }
703
704 /* pad with zeros */
705 for (i2=0; i2 < nx; i2++) {
706     for (i1=nt; i1 < nt1; i1++) {
707         tmp1[i2][i1] = 0.0f;
708         tmp2[i2][i1] = 0.0f;
709     }
710 }
711 for (i2=nx; i2 < nx2; i2++) {
712     for (i1=0; i1 < nt1; i1++) {
713         tmp1[i2][i1] = 0.0f;
714         tmp2[i2][i1] = 0.0f;
715     }
716 }
717 }
718
719 fft2_allocate(ctmp1);
720 fft2(tmp1[0],ctmp1);
721 for(ik=0; ik<nk;ik++){
722     ctmp1[ik] *=wf[ik];
723 }
724 ifft2(tmp1[0],ctmp1);
725 fft2_allocate(ctmp2);
726 fft2(tmp2[0],ctmp2);
727 for(ik=0; ik<nk;ik++){
728     ctmp2[ik] *=x[3*n+ik];
729 }
730 ifft2(tmp2[0],ctmp2);
731
732 for (i=0; i < n; i++) {
733     i1 = i%nt;
734     i2 = i/nt;
735
736     y[i] += s[i]*x[2*n+i] - x[i];
737     tmp1[i2][i1] = x[i];
738     tmp2[i2][i1] = x1[i];
739 }
740
741 /* pad with zeros */
742 for (i2=0; i2 < nx; i2++) {
743     for (i1=nt; i1 < nt1; i1++) {
744         tmp1[i2][i1] = 0.0f;
745         tmp2[i2][i1] = 0.0f;
746     }
747 }
748 for (i2=nx; i2 < nx2; i2++) {
749     for (i1=0; i1 < nt1; i1++) {
750         tmp1[i2][i1] = 0.0f;
751         tmp2[i2][i1] = 0.0f;
752     }
753 }
754 }
755
756 fft2_allocate(ctmp1);
757 fft2(tmp1[0],ctmp1);
758 for(ik=0; ik<nk;ik++){
759     ctmp1[ik] *=wf[ik];
760 }
761 ifft2(tmp1[0],ctmp1);
762 fft2_allocate(ctmp2);
763 fft2(tmp2[0],ctmp2);
764 for(ik=0; ik<nk;ik++){
765     ctmp2[ik] *=x[3*n+ik];
766 }
767 ifft2(tmp2[0],ctmp2);
768
769 for (i=0; i < n; i++) {
770     i1 = i%nt;
771     i2 = i/nt;
772
773     y[i] += s[i]*x[2*n+i] - x[i];
774     tmp1[i2][i1] = x[i];
775     tmp2[i2][i1] = x1[i];
776 }
777
778 /* pad with zeros */
779 for (i2=0; i2 < nx; i2++) {
780     for (i1=nt; i1 < nt1; i1++) {
781         tmp1[i2][i1] = 0.0f;
782         tmp2[i2][i1] = 0.0f;
783     }
784 }
785 for (i2=nx; i2 < nx2; i2++) {
786     for (i1=0; i1 < nt1; i1++) {
787         tmp1[i2][i1] = 0.0f;
788         tmp2[i2][i1] = 0.0f;
789     }
790 }
791 }
792
793 fft2_allocate(ctmp1);
794 fft2(tmp1[0],ctmp1);
795 for(ik=0; ik<nk;ik++){
796     ctmp1[ik] *=wf[ik];
797 }
798 ifft2(tmp1[0],ctmp1);
800 fft2_allocate(ctmp2);
801 fft2(tmp2[0],ctmp2);
802 for(ik=0; ik<nk;ik++){
803     ctmp2[ik] *=x[3*n+ik];
804 }
805 ifft2(tmp2[0],ctmp2);
806
807 for (i=0; i < n; i++) {
808     i1 = i%nt;
809     i2 = i/nt;
810
811     y[i] += s[i]*x[2*n+i] - x[i];
812     tmp1[i2][i1] = x[i];
813     tmp2[i2][i1] = x1[i];
814 }
815
816 /* pad with zeros */
817 for (i2=0; i2 < nx; i2++) {
818     for (i1=nt; i1 < nt1; i1++) {
819         tmp1[i2][i1] = 0.0f;
820         tmp2[i2][i1] = 0.0f;
821     }
822 }
823 for (i2=nx; i2 < nx2; i2++) {
824     for (i1=0; i1 < nt1; i1++) {
825         tmp1[i2][i1] = 0.0f;
826         tmp2[i2][i1] = 0.0f;
827     }
828 }
829 }
830
831 fft2_allocate(ctmp1);
832 fft2(tmp1[0],ctmp1);
833 for(ik=0; ik<nk;ik++){
834     ctmp1[ik] *=wf[ik];
835 }
836 ifft2(tmp1[0],ctmp1);
837 fft2_allocate(ctmp2);
838 fft2(tmp2[0],ctmp2);
839 for(ik=0; ik<nk;ik++){
840     ctmp2[ik] *=x[3*n+ik];
841 }
842 ifft2(tmp2[0],ctmp2);
843
844 for (i=0; i < n; i++) {
845     i1 = i%nt;
846     i2 = i/nt;
847
848     y[i] += s[i]*x[2*n+i] - x[i];
849     tmp1[i2][i1] = x[i];
850     tmp2[i2][i1] = x1[i];
851 }
852
853 /* pad with zeros */
854 for (i2=0; i2 < nx; i2++) {
855     for (i1=nt; i1 < nt1; i1++) {
856         tmp1[i2][i1] = 0.0f;
857         tmp2[i2][i1] = 0.0f;
858     }
859 }
860 for (i2=nx; i2 < nx2; i2++) {
861     for (i1=0; i1 < nt1; i1++) {
862         tmp1[i2][i1] = 0.0f;
863         tmp2[i2][i1] = 0.0f;
864     }
865 }
866 }
867
868 fft2_allocate(ctmp1);
869 fft2(tmp1[0],ctmp1);
870 for(ik=0; ik<nk;ik++){
871     ctmp1[ik] *=wf[ik];
872 }
873 ifft2(tmp1[0],ctmp1);
875 fft2_allocate(ctmp2);
876 fft2(tmp2[0],ctmp2);
877 for(ik=0; ik<nk;ik++){
878     ctmp2[ik] *=x[3*n+ik];
879 }
880 ifft2(tmp2[0],ctmp2);
881
882 for (i=0; i < n; i++) {
883     i1 = i%nt;
884     i2 = i/nt;
885
886     y[i] += s[i]*x[2*n+i] - x[i];
887     tmp1[i2][i1] = x[i];
888     tmp2[i2][i1] = x1[i];
889 }
890
891 /* pad with zeros */
892 for (i2=0; i2 < nx; i2++) {
893     for (i1=nt; i1 < nt1; i1++) {
894         tmp1[i2][i1] = 0.0f;
895         tmp2[i2][i1] = 0.0f;
896     }
897 }
898 for (i2=nx; i2 < nx2; i2++) {
899     for (i1=0; i1 < nt1; i1++) {
900         tmp1[i2][i1] = 0.0f;
901         tmp2[i2][i1] = 0.0f;
902     }
903 }
904 }
905
906 fft2_allocate(ctmp1);
907 fft2(tmp1[0],ctmp1);
908 for(ik=0; ik<nk;ik++){
909     ctmp1[ik] *=wf[ik];
910 }
911 ifft2(tmp1[0],ctmp1);
913 fft2_allocate(ctmp2);
914 fft2(tmp2[0],ctmp2);
915 for(ik=0; ik<nk;ik++){
916     ctmp2[ik] *=x[3*n+ik];
917 }
918 ifft2(tmp2[0],ctmp2);
919
920 for (i=0; i < n; i++) {
921     i1 = i%nt;
922     i2 = i/nt;
923
924     y[i] += s[i]*x[2*n+i] - x[i];
925     tmp1[i2][i1] = x[i];
926     tmp2[i2][i1] = x1[i];
927 }
928
929 /* pad with zeros */
930 for (i2=0; i2 < nx; i2++) {
931     for (i1=nt; i1 < nt1; i1++) {
932         tmp1[i2][i1] = 0.0f;
933         tmp2[i2][i1] = 0.0f;
934     }
935 }
936 for (i2=nx; i2 < nx2; i2++) {
937     for (i1=0; i1 < nt1; i1++) {
938         tmp1[i2][i1] = 0.0f;
939         tmp2[i2][i1] = 0.0f;
940     }
941 }
942 }
943
944 fft2_allocate(ctmp1);
945 fft2(tmp1[0],ctmp1);
946 for(ik=0; ik<nk;ik++){
947     ctmp1[ik] *=wf[ik];
948 }
949 ifft2(tmp1[0],ctmp1);
951 fft2_allocate(ctmp2);
952 fft2(tmp2[0],ctmp2);
953 for(ik=0; ik<nk;ik++){
954     ctmp2[ik] *=x[3*n+ik];
955 }
956 ifft2(tmp2[0],ctmp2);
957
958 for (i=0; i < n; i++) {
959     i1 = i%nt;
960     i2 = i/nt;
961
962     y[i] += s[i]*x[2*n+i] - x[i];
963     tmp1[i2][i1] = x[i];
964     tmp2[i2][i1] = x1[i];
965 }
966
967 /* pad with zeros */
968 for (i2=0; i2 < nx; i2++) {
969     for (i1=nt; i1 < nt1; i1++) {
970         tmp1[i2][i1] = 0.0f;
971         tmp2[i2][i1] = 0.0f;
972     }
973 }
974 for (i2=nx; i2 < nx2; i2++) {
975     for (i1=0; i1 < nt1; i1++) {
976         tmp1[i2][i1] = 0.0f;
977         tmp2[i2][i1] = 0.0f;
978     }
979 }
980 }
981
982 fft2_allocate(ctmp1);
983 fft2(tmp1[0],ctmp1);
984 for(ik=0; ik<nk;ik++){
985     ctmp1[ik] *=wf[ik];
986 }
987 ifft2(tmp1[0],ctmp1);
989 fft2_allocate(ctmp2);
990 fft2(tmp2[0],ctmp2);
991 for(ik=0; ik<nk;ik++){
992     ctmp2[ik] *=x[3*n+ik];
993 }
994 ifft2(tmp2[0],ctmp2);
995
996 for (i=0; i < n; i++) {
997     i1 = i%nt;
998     i2 = i/nt;
999
1000     y[i] += s[i]*x[2*n+i] - x[i];
1001     tmp1[i2][i1] = x[i];
1002     tmp2[i2][i1] = x1[i];
1003 }
1004
1005 /* pad with zeros */
1006 for (i2=0; i2 < nx; i2++) {
1007     for (i1=nt; i1 < nt1; i1++) {
1008         tmp1[i2][i1] = 0.0f;
1009         tmp2[i2][i1] = 0.0f;
1010     }
1011 }
1012 for (i2=nx; i2 < nx2; i2++) {
1013     for (i1=0; i1 < nt1; i1++) {
1014         tmp1[i2][i1] = 0.0f;
1015         tmp2[i2][i1] = 0.0f;
1016     }
1017 }
1018 }
1019
1020 fft2_allocate(ctmp1);
1021 fft2(tmp1[0],ctmp1);
1022 for(ik=0; ik<nk;ik++){
1023     ctmp1[ik] *=wf[ik];
1024 }
1025 ifft2(tmp1[0],ctmp1);
1027 fft2_allocate(ctmp2);
1028 fft2(tmp2[0],ctmp2);
1029 for(ik=0; ik<nk;ik++){
1030     ctmp2[ik] *=x[3*n+ik];
1031 }
1032 ifft2(tmp2[0],ctmp2);
1033
1034 for (i=0; i < n; i++) {
1035     i1 = i%nt;
1036     i2 = i/nt;
1037
1038     y[i] += s[i]*x[2*n+i] - x[i];
1039     tmp1[i2][i1] = x[i];
1040     tmp2[i2][i1] = x1[i];
1041 }
1042
1043 /* pad with zeros */
1044 for (i2=0; i2 < nx; i2++) {
1045     for (i1=nt; i1 < nt1; i1++) {
1046         tmp1[i2][i1] = 0.0f;
1047         tmp2[i2][i1] = 0.0f;
1048     }
1049 }
1050 for (i2=nx; i2 < nx2; i2++) {
1051     for (i1=0; i1 < nt1; i1++) {
1052         tmp1[i2][i1] = 0.0f;
1053         tmp2[i2][i1] = 0.0f;
1054     }
1055 }
1056 }
1057
1058 fft2_allocate(ctmp1);
1059 fft2(tmp1[0],ctmp1);
1060 for(ik=0; ik<nk;ik++){
1061     ctmp1[ik] *=wf[ik];
1062 }
1063 ifft2(tmp1[0],ctmp1);
1065 fft2_allocate(ctmp2);
1066 fft2(tmp2[0],ctmp2);
1067 for(ik=0; ik<nk;ik++){
1068     ctmp2[ik] *=x[3*n+ik];
1069 }
1070 ifft2(tmp2[0],ctmp2);
1071
1072 for (i=0; i < n; i++) {
1073     i1 = i%nt;
1074     i2 = i/nt;
1075
1076     y[i] += s[i]*x[2*n+i] - x[i];
1077     tmp1[i2][i1] = x[i];
1078     tmp2[i2][i1] = x1[i];
1079 }
1080
1081 /* pad with zeros */
1082 for (i2=0; i2 < nx; i2++) {
1083     for (i1=nt; i1 < nt1; i1++) {
1084         tmp1[i2][i1] = 0.0f;
1085         tmp2[i2][i1] = 0.0f;
1086     }
1087 }
1088 for (i2=nx; i2 < nx2; i2++) {
1089     for (i1=0; i1 < nt1; i1++) {
1090         tmp1[i2][i1] = 0.0f;
1091         tmp2[i2][i1] = 0.0f;
1092     }
1093 }
1094 }
1095
1096 fft2_allocate(ctmp1);
1097 fft2(tmp1[0],ctmp1);
1098 for(ik=0; ik<nk;ik++){
1099     ctmp1[ik] *=wf[ik];
1100 }
1101 ifft2(tmp1[0],ctmp1);
1103 fft2_allocate(ctmp2);
1104 fft2(tmp2[0],ctmp2);
1105 for(ik=0; ik<nk;ik++){
1106     ctmp2[ik] *=x[3*n+ik];
1107 }
1108 ifft2(tmp2[0],ctmp2);
1109
1110 for (i=0; i < n; i++) {
1111     i1 = i%nt;
1112     i2 = i/nt;
1113
1114     y[i] += s[i]*x[2*n+i] - x[i];
1115     tmp1[i2][i1] = x[i];
1116     tmp2[i2][i1] = x1[i];
1117 }
1118
1119 /* pad with zeros */
1120 for (i2=0; i2 < nx; i2++) {
1121     for (i1=nt; i1 < nt1; i1++) {
1122         tmp1[i2][i1] = 0.0f;
1123         tmp2[i2][i1] = 0.0f;
1124     }
1125 }
1126 for (i2=nx; i2 < nx2; i2++) {
1127     for (i1=0; i1 < nt1; i1++) {
1128         tmp1[i2][i1] = 0.0f;
1129         tmp2[i2][i1] = 0.0f;
1130     }
1131 }
1132 }
1133
1134 fft2_allocate(ctmp1);
1135 fft2(tmp1[0],ctmp1);
1136 for(ik=0; ik<nk;ik++){
1137     ctmp1[ik] *=wf[ik];
1138 }
1139 ifft2(tmp1[0],ctmp1);
1141 fft2_allocate(ctmp2);
1142 fft2(tmp2[0],ctmp2);
1143 for(ik=0; ik<nk;ik++){
1144     ctmp2[ik] *=x[3*n+ik];
1145 }
1146 ifft2(tmp2[0],ctmp2);
1147
1148 for (i=0; i < n; i++) {
1149     i1 = i%nt;
1150     i2 = i/nt;
1151
1152     y[i] += s[i]*x[2*n+i] - x[i];
1153     tmp1[i2][i1] = x[i];
1154     tmp2[i2][i1] = x1[i];
1155 }
1156
1157 /* pad with zeros */
1158 for (i2=0; i2 < nx; i2++) {
1159     for (i1=nt; i1 < nt1; i1++) {
1160         tmp1[i2][i1] = 0.0f;
1161         tmp2[i2][i1] = 0.0f;
1162     }
1163 }
1164 for (i2=nx; i2 < nx2; i2++) {
1165     for (i1=0; i1 < nt1; i1++) {
1166         tmp1[i2][i1] = 0.0f;
1167         tmp2[i2][i1] = 0.0f;
1168     }
1169 }
1170 }
1171
1172 fft2_allocate(ctmp1);
1173 fft2(tmp1[0],ctmp1);
1174 for(ik=0; ik<nk;ik++){
1175     ctmp1[ik] *=wf[ik];
1176 }
1177 ifft2(tmp1[0],ctmp1);
1179 fft2_allocate(ctmp2);
1180 fft2(tmp2[0],ctmp2);
1181 for(ik=0; ik<nk;ik++){
1182     ctmp2[ik] *=x[3*n+ik];
1183 }
1184 ifft2(tmp2[0],ctmp2);
1185
1186 for (i=0; i < n; i++) {
1187     i1 = i%nt;
1188     i2 = i/nt;
1189
1190     y[i] += s[i]*x[2*n+i] - x[i];
1191     tmp1[i2][i1] = x[i];
1192     tmp2[i2][i1] = x1[i];
1193 }
1194
1195 /* pad with zeros */
1196 for (i2=0; i2 < nx; i2++) {
1197     for (i1=nt; i1 < nt1; i1++) {
1198         tmp1[i2][i1] = 0.0f;
1199         tmp2[i2][i1] = 0.0f;
1200     }
1201 }
1202 for (i2=nx; i2 < nx2; i2++) {
1203     for (i1=0; i1 < nt1; i1++) {
1204         tmp1[i2][i1] = 0.0f;
1205         tmp2[i2][i1] = 0.0f;
1206     }
1207 }
1208 }
1209
1210 fft2_allocate(ctmp1);
1
```

```

281     y[n+i] += tmp1[i2][i1] + tmp2[i2][i1] - x[n+i];
282     y[2*n+i] += x2[i]*x[2*n+i] + w[i]*x[n+i];
283   }
284 }
285 }
```

Listing 3: chapter-lsrtm/code/Mtf2dprec.c

```

/* TF Weights Preconditioner for Real input as linear operator*/
/*
Copyright (C) 2004 University of Texas at Austin

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/
#include <math.h>
#include <rsf.h>
#include "tf2dprec.h"
#include "fft2.h"

int main(int argc, char* argv[])
{
    int nz, nx, nz2, nx2, nk, nzx, n_left, i;
    int nk_rfft, nx_rfft;
    bool adj;
    float *ww, *ff;
    float *pp, *qq;
    sf_file src, out, w, wf;

    sf_init(argc, argv);

    src = sf_input("in");
    out = sf_output("out");
    w = sf_input("w");
    wf = sf_input("wf");

    if (!sf_histint(src, "n1", &nz)) sf_error("No n1= in input");
    if (!sf_histint(src, "n2", &nx)) sf_error("No n2= in input");
    /* dim from frequency weight - derived from real fft2 */
    if (!sf_histint(wf, "n1", &nk_rfft)) sf_error("No n1= in wf");
    if (!sf_histint(wf, "n2", &nx_rfft)) sf_error("No n2= in wf");

    nzx = nz*nx;
    n_left = sf_leftsize(src, 2);

    nk = fft2_init(false, 1, nz, nx, &nz2, &nx2);

    if (nk_rfft*nx_rfft != nk) sf_error("FFT dimension error (nk)");
    if (nx_rfft != nx2) sf_error("FFT dimension error (nx)");
}
```

```

54
56     pp = sf_floatalloc(nzx);
58     qq = sf_floatalloc(nzx);
60     ww = sf_floatalloc(nz*nx);
62     sf_floatread(ww, nzx, w);
64     sf_fileclose(w);
66
68     ff = sf_floatalloc(nk);
70     sf_floatread(ff, nk, wf);
72     sf_fileclose(wf);
74
76     tf2dprec_init(nz, nx, nk, nz2, nx2, ww, ff);
78
80     sf_floatread(pp, nzx, src);
82
84     if (!sf_getbool("adj", &adj)) adj=false;
86
88     for (i=0; i < n_left; i++) {
90         if (adj) {
92             tf2dprec_lop(true, false, nzx, nzx, qq, pp);
94         } else {
96             tf2dprec_lop(false, false, nzx, nzx, pp, qq);
98         }
100    }
102
104    sf_floatwrite(qq, nzx, out);
106
108    exit(0);
110}

```

Listing 4: chapter-lsrtm/code/tf2dprec.c

```

/* TF Weights Preconditioner for Real input as linear oper. */
2 #include <rsf.h>
3 #include "tf2dprec.h"
4 #include "fft2.h"
5
6
7 static int nz, nx, nz2, nx2, nk;
8 static float *w, *wf, **tmp1;
9 static sf_complex *ctmp1; /* for 2D-fft */
10
11 void tf2dprec_init(int n1, /* trace length */
12                     int n2, /* number of length */
13                     int nk_in, /* total Fourier size [n1*n2] */
14                     int nz2_in, /* Fourier dim1 */
15                     int nx2_in, /* Fourier dim2 */
16                     float* ww /* [n1*n2] time weight */,
17                     float* ff /* [nk] frequency weight */)
18 /*< initialize >*/ {
19
20     nz = n1;
21     nx = n2;
22     nk = nk_in;
23     nz2 = nz2_in;
24     nx2 = nx2_in;
25     w = ww;
26

```

```

28     wf = ff;
29     /*for 2D fft*/
30     tmp1 = sf_floatalloc2(nz2, nx2);
31     ctmp1 = sf_complexalloc(nk);
32 }

34

36 void tf2dprec_close(void)
37 /*< clean allocated storage >/
38 {
39     free(*tmp1);
40     free(tmp1);
41     free(ctmp1);
42 }

44 void tf2dprec_lop(bool adj, bool add, int nxx, int nyy, float* x, float* y)
45 /*< linear operator >/
46 {

48     int i, i1, i2, ik;
49     if (nxx != nz*nx || nyy != nz*nx) sf_error("%s: Wrong size", __FILE__);
50
51     sf_adjnull(adj, add, nxx, nyy, x, y);

52     if (adj){ /* Adjoint*/
53
54         /* pad with zeros */
55         for (i2=0; i2 < nx; i2++) {
56             for (i1=0; i1 < nz; i1++) {
57                 i = i1+i2*nz;
58                 tmp1[i2][i1] = w[i]*y[i];
59             }
60             for (i1=nz; i1 < nz2; i1++) {
61                 tmp1[i2][i1] = 0.0f;
62             }
63         }
64         for (i2=nx; i2 < nx2; i2++) {
65             for (i1=0; i1 < nz2; i1++) {
66                 tmp1[i2][i1] = 0.0f;
67             }
68         }
69
70         /* forward FFT */
71         fft2_allocate(ctmp1);
72         fft2(tmp1[0], ctmp1);

74         /* frequency weight */
75         for (ik=0; ik < nk; ik++) {
76             ctmp1[ik] *= wf[ik];
77         }
78         /* inverse FFT */
79         ifft2(tmp1[0], ctmp1);

82         for (i=0; i < nz*nx; i++) {
83             i1 = i%nz;
84             i2 = i/nz;
85             x[i] += tmp1[i2][i1];
86         }
87     } else{ /* Forward */
88

```

```

90      /* pad with zeros */
91      for (i2=0; i2 < nx; i2++) {
92          for (i1=0; i1 < nz; i1++) {
93              i = i1+i2*nz;
94              tmp1[i2][i1] = x[i];
95          }
96          for (i1=nz; i1 < nz2; i1++) {
97              tmp1[i2][i1] = 0.0f;
98          }
99      }
100
101      for (i2=nx; i2 < nx2; i2++) {
102          for (i1=0; i1 < nz2; i1++) {
103              tmp1[i2][i1] = 0.0f;
104          }
105      }
106
107      /* forward FFT */
108      fft2_allocate(ctmp1);
109      fft2(tmp1[0], ctmp1);
110
111      /* frequency weight */
112      for (ik=0; ik < nk; ik++) {
113          ctmp1[ik] *= wf[ik];
114      }
115      /* inverse FFT */
116      ifft2(tmp1[0], ctmp1);
117
118      for (i=0; i < nz*nx; i++) {
119          i1 = i%nz;
120          i2 = i/nz;
121          y[i] += w[i]*tmp1[i2][i1];
122      }
123  }

```

Bibliography

- Aoki, N., and G. T. Schuster, 2009, Fast least-squares migration with a deblurring filter: *Geophysics*, **74**, WCA83–WCA93.
- Bleistein, N., 1987, On the imaging of reflectors in the earth: *Geophysics*, **52**, 931–942.
- Claerbout, J. F., 1992, Earth soundings analysis: Processing versus inversion, **6**.
- Dai, W., P. Fowler, and G. T. Schuster, 2012, Multi-source least-squares reverse time migration: *Geophysical Prospecting*, **60**, 681–695.
- Fomel, S., 2007a, Local seismic attributes: *Geophysics*, **72**, A29–A33.
- , 2007b, Shaping regularization in geophysical-estimation problems: *Geophysics*, **72**, R29–R36.
- Fomel, S., P. Sava, I. Vlad, Y. Liu, and V. Bashkardin, 2013, Madagascar: open-source software project for multidimensional data analysis and reproducible computational experiments: *Journal of Open Research Software*, **1**, e8.
- Greer, S., Z. Xue, and S. Fomel, 2018, Improving migration resolution by approximating the least-squares hessian using nonstationary amplitude and frequency matching, *in* SEG Technical Program Expanded Abstracts 2018: Society of Exploration Geophysicists, 4261–4265.
- Guitton, A., 2004, Amplitude and kinematic corrections of migrated images for nonunitary imaging operators: *Geophysics*, **69**, 1017–1024.
- Hou, J., and W. W. Symes, 2015, An approximate inverse to the extended born modeling operator an approximate inverse operator: *Geophysics*, **80**, R331–R349.

- , 2016, Accelerating extended least-squares migration with weighted conjugate gradient iteration: *Geophysics*, **81**, S165–S179.
- Hu, J., G. T. Schuster, and P. A. Valasek, 2001, Poststack migration deconvolution: *Geophysics*, **66**, 939–952.
- Jones, I., 2014, Tutorial: migration imaging conditions: First break, **32**.
- Kaur, H., N. Pham, and S. Fomel, 2020, Improving resolution of migrated images by approximating the inverse hessian using deep learning: *Geophysics*, **85**, 1–62.
- Keys, R. G., and D. J. Foster, 1998, Comparison of seismic inversion methods on a single real data set: Society of Exploration Geophysicists.
- Liner, C. L., and R. G. Clapp, 2004, Nonlinear pairwise alignment of seismic traces: *Geophysics*, **69**, 1552–1559.
- Miller, D., M. Oristaglio, and G. Beylkin, 1987, A new slant on seismic imaging: Migration and integral geometry: *Geophysics*, **52**, 943–964.
- Nemeth, T., C. Wu, and G. T. Schuster, 1999, Least-squares migration of incomplete reflection data: *Geophysics*, **64**, 208–221.
- Pica, A., J. Diet, and A. Tarantola, 1990, Nonlinear inversion of seismic reflection data in a laterally invariant medium: *Geophysics*, **55**, 284–292.
- Rickett, J. E., 2003, Illumination-based normalization for wave-equation depth migration: *Geophysics*, **68**, 1371–1379.
- Ronen, S., and C. L. Liner, 2000, Least-squares dmo and migration: *Geophysics*, **65**, 1364–1371.
- Sun, J., S. Fomel, T. Zhu, and J. Hu, 2016, Q-compensated least-squares reverse time migration using low-rank one-step wave extrapolation: *Geophysics*, **81**, S271–S279.
- Tarantola, A., 1984a, Inversion of seismic reflection data in the acoustic approximation: *Geophysics*, **49**, 1259–1266.

- , 1984b, Linearized inversion of seismic reflection data: Geophysical prospecting, **32**, 998–1015.
- Versteeg, R., 1994, The marmousi experience: Velocity model determination on a synthetic complex data set: The Leading Edge, **13**, 927–936.
- Wang, P., A. Gomes, Z. Zhang, and M. Wang, 2016, Least-squares rtm: Reality and possibilities for subsalt imaging, *in* SEG Technical Program Expanded Abstracts 2016: Society of Exploration Geophysicists, 4204–4209.
- Wiggins, R. A., 1972, The general linear inverse problem: Implication of surface waves and free oscillations for earth structure: Reviews of Geophysics, **10**, 251–285.
- Wong, M., S. Ronen, and B. Biondi, 2011, Least-squares reverse time migration/inversion for ocean bottom data: A case study, *in* SEG Technical Program Expanded Abstracts 2011: Society of Exploration Geophysicists, 2369–2373.
- Xue, Z., Y. Chen, S. Fomel, and J. Sun, 2016, Seismic imaging of incomplete data and simultaneous-source data using least-squares reverse time migration with shaping regularization: Geophysics, **81**, S11–S20.
- Yu, J., J. Hu, G. T. Schuster, and R. Estill, 2006, Prestack migration deconvolution: Geophysics, **71**, S53–S62.

Vita

Tharit Tangkijwanichakul graduated from Triam Udom Suksa High School in Thailand. He received a scholarship from PTT Exploration and Production (PT-TEP), Thailand state-owned E&P Company, to study exploration geophysics in the USA. He pursued a Bachelor of Science degree in Geophysics from the Jackson School of Geosciences along with getting the certificate in Computational Science and Engineering from Oden Institute of Computational Science and Engineering. After graduation, he will go back to Thailand and work full-time as a geophysicist for PTTEP.

Permanent address: 908 Poplar St. Austin, TX USA 78705

This thesis was typeset with \LaTeX^{\dagger} by the author.

^{\dagger} \LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's \TeX Program.