Preface

This thesis has been submitted to meet the partial requirements for obtaining the Ph.D. degree from the Technical University of Denmark (DTU). The study was carried out from October 2000 to November 2003 according to the Ph.D. Programme in Mathematics, at Informatics and Mathematical Modelling (IMM), under the supervision of Professor Per Christian Hansen.

Acknowledgments

I wish to thank my supervisor Per Christian Hansen for his encouragement, interest and help the project. I also wish to thank Professor James G. Nagy and everybody else at, and around, Emory University during my visit in the spring of 2002 for providing great inspiration and a pleasant stay.

Further, I wish to thank my fellow Ph.D. students and colleagues at IMM for providing diversion during lunch time, coffee time and work time. I also wish to thank the students that have helped me find bugs and unexpected features in my programs. Finally, I wish to thank my family and friends for making me relax when off duty.

Kgs. Lyngby, May 11, 2004

Michael Jacobsen

Resumé

Klassen af inverse "ill-posed" problemer beskrives sammen med en række numeriske standard værktøjer og basale begreber fra lineær algebra, statistik og optimering.

Kendte algoritmer til løsning af lineære inverse ill-posed problemer analyseres med henblik på, hvorledes de kan opdeles i moduler. Disse moduler kombineres derefter for at danne nye regulariserings algoritmer med andre egenskaber end dem de tog udgangspunkt i. Flere nye varianter bliver afprøvet med Matlab toolboxen M$\mathcal{OO}$Re Tools, der er udviklet i forbindelse med denne afhandling.

Objekt orienterede programmeringsteknikker anvendes til at abstrahere selve det inverse problem og dets data. Herved kan der skrives regulariserings-algoritmer, der automatisk udnytter struktur i det inverse problem uden en decideret omskrivning af algoritmen.

Et stopkriterium i forbindelse med Lanczos bidiagonalisering er forklaret. Stopkriteriet kan anvendes i forbindelse med parametervalgs-metoder. Det undersøges, hvordan standard-form transformationen kan overføres til den objekt orienterede pakke. En prækonditioner med sigte på Tikhonov regularisering i generel form forklares og der udføres eksperimenter, som demonstrerer simpliciteten og effektiviteten af prækonditioneren.

Via en tutorial for M$\mathcal{OO}$Re Tools er det vist, hvordan flere af afhandlingens figurer er skabt. Den inkluderede artikel "Subspace Preconditioned LSQR for Ill-Posed Problems" diskuterer en algoritme, der ikke direkte kan implementeres med M$\mathcal{OO}$Re Tools.

Abstract

The class of linear ill-posed problems is introduced along with a range of standard numerical tools and basic concepts from linear algebra, statistics and optimization.

Known algorithms for solving linear inverse ill-posed problems are analyzed to determine how they can be decomposed into independent modules. These modules are then combined to form new regularization algorithms with other properties than those we started out with. Several variations are tested using the Matlab toolbox M$\mathcal{OO}$Re Tools created in connection with this thesis.

Object oriented programming techniques are explained and used to set up the ill-posed problems in the toolbox. Hereby, we are able to write regularization algorithms that automatically exploit structure in the ill-posed problem without being rewritten explicitly.

We explain how to implement a stopping criteria for a parameter choice method based upon an iterative method. The parameter choice method is also used to demonstrate the implementation of the standard-form transformation. We have implemented a simple preconditioner aimed at the preconditioning of the general-form Tikhonov problem and demonstrate its simplicity and efficiency.

The steps taken with M$\mathcal{OO}$Re Tools to produce several of the figures are demonstrated in the toolbox tutorial. We have included the article "Subspace Preconditioned LSQR for Ill-Posed Problems" that discusses an algorithm that is not easily implemented with M$\mathcal{OO}$Re Tools.

Notation

Chapter 2 includes further details on notation, decompositions and other mathematical tools.

A matrix is denoted with uppercase bold roman letters like $\mathbf{A}$. The ith column of a matrix $\mathbf{A}$ is written $\mathbf{a}_i$ (it can be seen as a vector). Diagonal matrices are written with upper case Greek letters. The diagonal elements are written with lower case Greek letters with index, that is, σ_i is the ith diagonal element of $\mathbf{\Sigma}$. An block in a block matrix $\mathbf{A}$ is denoted like $\mathbf{A}_{11}$, that is, bold with indices.

Vectors are written with lowercase bold letters such as $\mathbf{a}$ and $\mathbf{v}$. The ith element of a vector $\mathbf{a}$ is denoted a_i (as it is a scalar). The notation $[\cdot]_i$ used to denote the ith element of the vector inside the brackets. A sub-vector is denoted $\mathbf{a}_1$, that is, still bold. Occasionally we will use a Matlab like notation where, for example, $\mathbf{A}_{:,1:k}$ is the first k columns of the matrix $\mathbf{A}$. Scalars are written with non-bold roman or Greek letters. We have reserved a number of letters for specific and much used entities.

Symbol	Description
$\mathbf{A}$	Square normal equation matrix (e.g., $\mathbf{K}^T\mathbf{K}$)
$\mathbf{b}$	Normal equation right hand side (e.g., $\mathbf{K}^T\mathbf{y}$)
$\mathbf{e}_k$	Vector of appropriate length with a "1" at the kth position
δ_{ij}	Dirac's delta
$\mathbf{K}$	Transfer matrix ($m \times n$)
$\mathbf{K}^\dagger$	Moore-Penrose pseudo-inverse of $\mathbf{K}$
$\mathbf{L}_{\mathbf{K}}^\dagger$	$\mathbf{K}$-weighted pseudo-inverse of $\mathbf{L}$
$\mathbf{K}_\lambda^\#$	Regularized inverse of $\mathbf{K}$
$\mathbf{x}$	Solution vector
$\mathbf{x}_\lambda$	Regularized solution with regularization parameter λ
$\mathbf{y}$	Noisy right hand side
$\mathbf{L}$	Regularization matrix

$\mathbf{L}_1$	Approximation to first derivative
$\mathbf{L}_2$	Approximation to second derivative
$\mathcal{L}(\cdot, \cdot)$	The Lagrangian function
$\odot$	Element-wise multiplication (Hadamard product)
$\otimes$	Kronecker product
∇f	Gradient of f
$H(f)$	Hessian of f
$a \leftarrow b$	Assign b to a

Contents

Notation **vii**

1 Introduction **1**
1.1 Inverse Problems . . . 1
1.1.1 Ill-Posed Problems . . . 2
1.1.2 Regularization and Modules . . . 4
1.2 Object Oriented Programming . . . 4
1.3 Outline of the Thesis . . . 5

2 Mathematical Tools **7**
2.1 The Continuous Ill-Posed Problem . . . 7
2.2 Matrix Decompositions . . . 11
2.3 Krylov Subspace Methods . . . 14
2.4 The Givens Transformation . . . 19
2.5 Robust Estimation . . . 20
2.5.1 Robust Penalty Functions . . . 23
2.6 Optimization and Optimality . . . 24

3 Modular Algorithms **29**
3.1 Two Regularization Classes . . . 29
3.1.1 Penalty Methods . . . 30
3.1.2 Projection Methods . . . 32
3.1.3 Hybrid Methods . . . 34
3.2 Modules . . . 35
3.2.1 Penalty Methods . . . 35
3.2.2 Projection Methods . . . 39
3.2.3 Hybrid Methods . . . 41
3.3 Parameter-Choice Methods . . . 41
3.4 Combining Modules . . . 45

3.4.1 Combining Modules — Numerical Experiments 46
3.4.2 PP-LSQR 46
3.4.3 A GMRES-Tikhonov Hybrid Method 46
3.4.4 Robust TSVD 47
3.4.5 A PP-LSQR-TSVD Hybrid Method 49
3.4.6 A Robust Tikhonov Method 50
3.4.7 L-Curves for the Fair Problem 51
3.5 Summary 53

4 Object Oriented Programming 55
4.1 Programming Paradigms 55
4.1.1 Object Oriented Programming 57
4.2 Object Oriented Matlab 58
4.3 Regularization and Objects 60
4.3.1 Base Classes 61
4.3.2 Utility Classes 64
4.4 Use of Classes and Operator Overloading 67
4.5 Case Study: The Kronecker Product 69
4.5.1 A Kronecker Product 69
4.6 Algorithmic Implications 77
4.6.1 Subspace Preconditioned LSQR 78
4.7 Performance Implications 78
4.8 Summary 79

5 New Techniques 81
5.1 Large-Scale Parameter Choice 82
5.2 The **K**-weighted Pseudo-Inverse 91
5.2.1 Evaluation of the Weighted Pseudo-Inverse 94
5.3 A Tikhonov Preconditioner 96
5.3.1 The General Case 99
5.3.2 Numerical Experiments 102
5.4 Summary 108

6 Conclusion 111
6.1 Modular Algorithms 111
6.2 Object Oriented Programming 112
6.3 Other Results 113
6.4 The Toolbox 113
6.5 Further Work 114

A Toolbox Tutorial 119
A.1 Installation and Setup 119
A.2 Creating a Test-Problem 120
A.2.1 Tikhonov and LSQR 121
A.2.2 Getting Help 122

A.3 Thesis Examples 123
A.3.1 Writing a new LinearOperator child 129
A.3.2 Writing a new Vector child 130
A.3.3 Using the Option Structure 131

B MOORe Tools Contents **133**

C Validation of Toolbox **137**
C.1 Testing the Vector hierarchy 139
C.1.1 Test of Vector 141
C.2 Testing the LinearOperator hierarchy 143
C.2.1 Test of DiagonalOperator 145
C.2.2 Test of IdentityOperator 146
C.2.3 Test of KroneckerProduct 147
C.2.4 Test of KroneckerProduct2D and KroneckerProduct3D 148
C.2.5 Test of Matrix 150
C.2.6 Test of NullOperator 151
C.2.7 Test of OperatorArray 152
C.2.8 Test of OperatorProduct 152
C.2.9 Test of OperatorSum 153
C.2.10 Test of OperatorSVD 154
C.2.11 Test of PermuationOperator 154
C.2.12 Test of SparseMatrix 155
C.2.13 Test of TikhPrecond 156
C.2.14 Test of VectorCollection 156
C.2.15 Test of VectorReshape 157
C.2.16 Test of WeightedPseudoInverse 157
C.3 Testing the Algorithms 158
C.3.1 Test of Parameter Choice Methods 167
C.3.2 Test of Support Functions 169
C.3.3 The Combined Algorithms 171

D New Test Problems **173**
D.1 Steady-State Heat Distribution 173
D.2 Interpolation 176
D.3 Deblurring Problems 176
D.4 Gravity 178

E Subspace Preconditioned LSQR for Discrete Ill-Posed Problems **179**

Bibliography **197**

Chapter 1

Introduction

For a long time mathematicians felt that ill-posed problems cannot describe real phenomena and objects. However, we shall show in the present book that the class of ill-posed problems includes many classical mathematical problems and, most significantly, that such problems have important applications. Tikhonov and Arsenin, "Solutions of Ill-Posed Problems" [123].

Give a digital computer a problem in arithmetic, and it will grind away methodically, tirelessly, at gigahertz speed, until ultimately it produces the wrong answer. … Even an innocent-looking number like 1/10 can cause no end of trouble: In most cases, the computer cannot even read it in or print it out exactly, much less perform exact calculations with it. Brian Hayes [74].

Algorithms are created to solve problems of some kind. The algorithms and methods described in this thesis deal with problems that, in a sense, are unsolvable or at least very hard to solve. However, using various "tricks" we are able to find, not the exact solution, but an approximate solution that is hopefully good enough to be used. The problems we will treat are *inverse* and *ill-posed* problems.

1.1 Inverse Problems

How is inverse to be understood? The usual physical equations describe how some premises or causes result in something that can be observed. Think of a planetary system with positions and velocities of all objects. On the basis of the premises we can compute the trajectories of the planets and how an observation would be at some time in the future. The inverse problem would be to find the premises, that is, the velocities and positions at an earlier time from this observation. The forward problem can be solved by an ordinary differential equation solver and so can the inverse by using negative time steps. However, it is not always that easy.

Consider a very simple mathematical example. If you know (the premises) x_1 and x_2 you easily observe their sum $x_1 + x_2$ — the forward problem. The inverse problem is, given the sum, to find the causes or premises. In our very simple example we see that we face a problem. For example, if the sum is 10 what are x_1 and x_2? What if the observation is inexact and contaminated with noise? This example is maybe the most simple linear ill-posed inverse problem we can think of.

The previous example of an inverse problem might seem irrelevant and of no practical use. However, inverse problems appear in a wide range of applications, for example medical tomography [102], geophysics [37], sound source detection [117] and image restoration [10]. In [53] and [54] Groetsch lists a number of illustrative examples of the diversity of inverse problems. The journal "Inverse Problems" (IOP Publishing Ltd.) is also full of different examples of more or less important real world inverse problems.

In this thesis we will look at discrete inverse linear ill-posed problems, with the formulation

$$\mathbf{K}\mathbf{x} = \mathbf{y},$$

where $\mathbf{x}$ is the unknown (the premises), $\mathbf{K}$ represents the system and $\mathbf{y}$ the observations. The sum example fits into this framework. Note, that we do not cover the more general non-linear inverse problems under which the planetary problem falls. See e.g. [36, 99, 124] for theory about the non-linear inverse problems.

Figure 1.1 illustrates the atmospheric blur problem. Astronomers view the sky through not only their telescopes but also the atmosphere. The forward problem models how the light passes through the atmosphere (via $\mathbf{K}_1$) and telescope ($\mathbf{K}_2$) before it is measured giving $\mathbf{x}$. The telescopes can be made such that $\mathbf{K}_2 \approx \mathbf{I}$ leaving $\mathbf{K}_1$, the atmosphere, as the source of problems. Most observatories are built on mountains to lessen the effect of the atmosphere, and in the case of the Hubble space telescope the influence of the atmosphere was removed by moving the telescope outside the atmosphere and into space, that is, effectively setting $\mathbf{K}_1 = \mathbf{I}$. However, from a inverse problem point of view a somewhat funny but quite expensive story is linked with the Hubble space telescope! When it was launched and put into orbit the telescope had a defect in its optical system which caused another inverse problem because $\mathbf{K}_2$ was not close to the identity as intended. Until a repair was made three years later mathematics was used to fix the error. See [2] for the full story.

1.1.1 Ill-Posed Problems

Ill-posed problems violate one or more of the conditions for a well-posed problem as defined by Hadamard [56]

1. the solution exists,
2. the solution is unique,

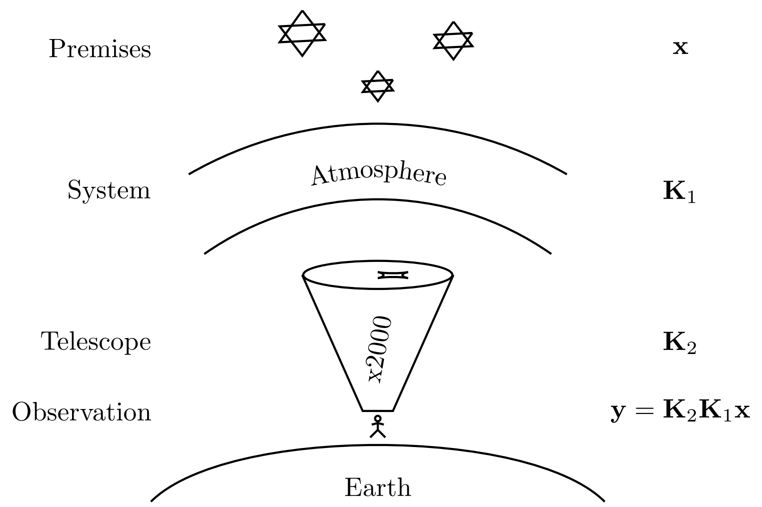


Figure 1.1: The stars emit light $\mathbf{x}$ which is scattered by the atmosphere $\mathbf{K}_1\mathbf{x}$ before it goes through the telescope $\mathbf{K}_2\mathbf{K}_1\mathbf{x}$ and is finally detected by an eye or a camera ($\mathbf{y} = \mathbf{K}_2\mathbf{K}_1\mathbf{x}$). The scattering process $\mathbf{K}_1$ and the path through the telescope $\mathbf{K}_2$ form the forward problem. The inverse problem is to find out how the photons arrived before they entered the atmosphere, that is, to find $\mathbf{x}$ from the observation $\mathbf{y}$.

3. the solution depends continuously on the exact right hand side.

Discrete linear problems violate condition 1 if the problem is rank-deficient and the right-hand side has a component in the null-space of the $\mathbf{K}$. If the operator is rank-deficient condition two is also violated as any component in the null-space may be added to a solution and still produce a solution. By choosing to solve for the least squares solution of minimum norm we avoid these problems. For a discrete inverse problem $\mathbf{K}\mathbf{x} = \mathbf{y}$ with the least squares solution of minimum norm

$$\mathbf{x} = \mathbf{K}^{\dagger}\mathbf{y},$$

we see that none of the conditions are in fact ever violated. However, if the continuous problem from which we have derived the problem violates condition 3 we still get an unwanted behavior. Characteristic for discrete ill-posed problems is that even very small changes in $\mathbf{y}$ compared to the exact values may destroy any attempt to solve for $\mathbf{x}$ with standard methods like Gaussian elimination or QR factorization. These changes can come from noise and limited accuracy in measurement devices or even from the limited number of digits in the computer's floating point representation. If we want to extract useful information from our system we need to use so-called regularization methods.

1.1.2 Regularization and Modules

A regularization method stabilizes the solution by adding constraints to the system and thereby reduces the influence of errors and noise. The art of regularization is to apply the right kind and the right amount of regularization. Secondly it is important to select the right algorithm for solving the regularized problem.

Already when the problem is discretized we regularize the problem. The problem is not ill-posed in a strict mathematical sense as discussed in the previous section (and next chapter). However, we will only work with already discretized problems.

Regularization routines come in many different colors and shapes each optimized, to some degree, toward a specific goal. We will try to extract common parts and develop *modular* algorithms that can be combined. Thereby we hope that it is possible to test the problem at hand with several different regularization methods to decide which combination of modules fits the problem and knowledge at hand. We will also see how the modular framework enables us to experiment with non-standard regularization methods. The aim is not to compare the algorithms extensively but to make it easy to compare them.

The package Regularization Tools [68] is a package with several algorithms aimed at ill-posed inverse problems. The package is aimed at experimenting and solving fairly small problems that can be expressed with full matrices. This thesis is accompanied by the Matlab package M$\mathcal{OO}$Re Tools, Modular $\mathcal{O}$bject $\mathcal{O}$oriented Regularization Tools, that demonstrates the modular framework described in this thesis. The package is available from the Internet at the location

`http://www.imm.dtu.dk/nag/software`

1.2 Object Oriented Programming

Making a computer do what you want it to do is hard. Especially because the computer is very picky, doing only what it is told to do and nothing else. In the early years of computing you had to control where every byte went and programming even simple things was tedious. Programming languages like Fortran eased the task considerably by introducing constructs that enabled the programmer to decompose the code into small manageable and sometimes reusable chunks. The library LAPACK [3] is an example of such a set of subroutines that are well tested and can be reused making development of sophisticated programs much easier.

The next step was to introduce objects and classes which hide data and implementation and only expose an interface. Hiding the facts about implementation and only exposing an interface increases the chance that modifications made in one part do not introduce bugs in other parts.

We will use object oriented techniques to exploit structures in the problem operator. Algorithms will use operators according to a specified interface. Then it is the job of the programmer to implement this interface as efficiently as possible. Without the object oriented approach it has often been necessary to rewrite the actual algorithm to take advantage of problem specific structures.

1.3 Outline of the Thesis

This first chapter has given a brief overview of what an ill-posed problem is and general methodologies to solve problems. The objectives of our software package have briefly been explained.

In Chapter 2 we first elaborate on the properties of an inverse ill-posed problem. We start with the continuous problem and then show how also a discretized problem has unfortunate properties even though it is not ill-posed in the sense of Hadamard. Then we survey some basic mathematical topics, and notation is defined in more detail. We define the SVD and GSVD decompositions, and discuss three algorithms, that form the basis of many iterative methods. We also briefly discuss robust estimation and state a vital result on optimization with equality constraints.

Chapter 3 explains how we decompose some of the well-known regularization algorithms into modules. We divide the regularization algorithms into three groups. For each group of methods we find a general set of solution methods which we call modules. These modules can then be combined to form other regularization schemes where we can tailor the properties to the problem at hand. Numerical experiments are used to illustrate the many different regularization methods that can be created from combining the modules.

In Chapter 4 we explain how object oriented techniques can be applied to the linear algebra and optimization algorithms that we used to construct the modules. We discuss where object oriented techniques are introduced and to what purpose in a general context. Then we use the object oriented approach with the linear algebra we face in our regularization methods. We use the two-term Kronecker product as a case study to illustrate how the objects and algorithms interplay.

Chapter 5 goes through some of the new ideas developed during the creation of the package. We look at a stopping criteria for a group of parameter choice methods. Then we consider the standard-form transformation used to simplify a problem, and finally we propose a simple preconditioner. The ideas are also illustrated with numerical experiments.

Finally in Chapter 6 we sum up the results and conclusions that can be made from this work and point to areas where further research seems appropriate. We also discuss how and where improvements to the accompanying package M$\mathcal{OO}$Re Tools can be done.

A tutorial for the M$\mathcal{OO}$Re Tools package is included in App. A. The tutorial shows the steps used to create several of the plots in this thesis and it details how to implement new problems with the object oriented approach. Lists of the available functions provided with the package are found in App. B. In App. C we show the steps taken to validate the correctness of the code in the M$\mathcal{OO}$Re Tools package, and in App. D we have included a review of the new test problems introduced.

Lastly in App. E, the paper “Subspace Preconditioned LSQR for Ill-Posed Problems” authored by Michael Jacobsen, Per Christian Hansen and Michael A. Saunders [71] is included as it is used in the discussion of problems with the object oriented approach.

Chapter 2

Mathematical Tools

While mathematicians, as a group, are not known for their fearlessness, just what is it about (2.6) that could strike terror in their hearts? Groetsch, "Inverse Problems in the Mathematical Sciences" [53, p. 36], about the innocent looking Fredholm integral equation of the first kind seen in (2.1) below.

What do we mean by "infinity"? On an informal, intuitive level, the main feature is that it's big. Very big. No, a lot bigger than that. Bigger than you imagine. Bigger than you can imagine. Ian Stewart, "Never Ending Story" [120].

In this chapter we briefly survey some of the tools and concepts needed later. As a consequence this chapter also defines some of the notation in more detail (see also the Notation pages in the preamble of this thesis). We will start with a discussion of the continuous problem in the form of the classic Fredholm equation of the first kind and then move on to the tools of trade for the discrete finite dimensional problem. Optimization and optimality conditions will be explained for our special type of problem and the concept of "robust estimation" will be introduced and justified.

2.1 The Continuous Ill-Posed Problem

The following discussion extracts the essentials of the continuous ill-posed problem. See, for example, Engl et al. [36] for a much more in-depth treatment of this subject. The Fredholm integral equation of the first kind with a square integrable kernel $k(s,t)$ can always be written in the generic form

$$\int_0^1 k(s,t)f(t)dt = g(s), \quad 0 \le s \le 1. \tag{2.1}$$

In the inverse problem the kernel $k(s,t)$ and the right-hand side $g(s)$ are the known functions and $f(t)$ is the unknown solution. We will also write (2.1) as $Kf = g$. Equation (2.1) seems harmless at first sight but we will see that severe problems can appear when solving for f—and even "strike terror in the hearts of mathematicians?"

We required the kernel $k(s,t)$ to be square integrable, that is,

$$\|k(s,t)\|^2 = \int_0^1 \int_0^1 k(s,t) ds dt$$

is bounded. Then we can use the singular value expansion (SVE) to write $k(s,t)$ as an infinite sum

$$k(s,t) = \sum_{i=1}^{\infty} \sigma_i u_i(s) v_i(t), \tag{2.2}$$

where $u_i(s)$ and $v_i(t)$ are the singular functions of K. The singular functions are orthonormal, that is,

$$\langle u_i, u_j \rangle = \langle v_i, v_j \rangle = \delta_{ij}$$

where $\langle f, g \rangle$ is the usual inner product $\langle f, g \rangle = \int_0^1 f(t)g(t)dt$ and δ_{ij} is the usual Dirac's delta. The singular values σ_i are non-negative and in non-increasing order,

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq 0.$$

The function f can be written in terms of the singular functions v_i and the orthogonal projection Q_v onto $\mathcal{R}(K^*)^{\perp}$. Similarly g can be written in terms of u_i and the orthogonal projection Q_u onto $\mathcal{R}(K)^{\perp}$. That is,

$$f = \sum_{i=1}^{\infty} \langle v_i, f \rangle v_i + Q_v f \tag{2.3}$$

$$g = \sum_{i=1}^{\infty} \langle u_i, g \rangle u_i + Q_u g. \tag{2.4}$$

A solution only exists if $g \in \mathcal{R}(K)$ or equivalently $Q_u g = 0$. Otherwise, using the right-hand side projected into $\mathcal{R}(K)$, that is, $\tilde{g} = \sum_{i=1}^{\infty} \langle u_i, g \rangle u_i$ produces a best-approximate solutions also called least squares solutions, cf. [36]. Furthermore, if $Q_v \neq 0$ we do not have a unique solution because $f_0 + Q_v f_1$ is also a solution if $f_0 \in \mathcal{N}(K)^{\perp}$ is the particular solution and f_1 is any arbitrary function. If we select $f_1 = 0$ we get the minimum norm least squares solution.

To find the least squares solution of minimum norm we insert the sums (2.2), (2.3) (without $Q_v f$) and (2.4) (without $Q_u g$) into the Fredholm integral equation (2.1) giving

$$\begin{aligned}\int_0^1 \left(\sum_{i=1}^{\infty} \sigma_i u_i v_i(t)\right) f(t)dt &= \sum_{i=1}^{\infty} \left(\sigma_i u_i \int_0^1 v_i(t) f(t) dt\right) \\ &= \sum_{i=1}^{\infty} \sigma_i \langle v_i, f\rangle u_i \\ &= \sum_{i=1}^{\infty} \langle u_i, g\rangle u_i .\end{aligned}$$

For all $\sigma_j > 0$ we form the inner product with u_j on each side to get the coefficient for each term in the expansion (2.3)

$$\begin{gathered}\left\langle u_j, \sum_{i=1}^{\infty} \sigma_i \langle v_i, f\rangle u_i \right\rangle = \left\langle u_j, \sum_{i=1}^{\infty} (u_i, g) u_i \right\rangle \\ \Updownarrow \\ \sigma_j \langle v_j, f\rangle = \langle u_j, g\rangle \\ \Updownarrow \\ \langle v_j, f\rangle = \frac{\langle u_j, g\rangle}{\sigma_j} \quad \forall \sigma_j > 0.\end{gathered}$$

We can now write a formula for the least squares solution of minimum norm in terms of the SVE

$$K^{\dagger} g = \sum_{\sigma_i > 0} \frac{\langle u_i, g\rangle}{\sigma_i} v_i, \tag{2.5}$$

where the operator $K^{\dagger}$ is called the Moore-Penrose pseudo-inverse (which we will meet again later in its discrete form). The coefficient to the solution component v_i includes a division by σ_i that is approaching zero when the index increases. From here on we assume that $\dim \mathcal{R}(K) = \infty$, that is, K is compact, all singular values are non-zero and $\lim_{\imath \to \infty} \sigma_i = 0$. To get a finite solution, that is $\|K^{\dagger} g\|_2^2 < \infty$, we find with Parseval's equation that

$$\|K^{\dagger} g\|_2^2 = \left\| \sum_{i=1}^{\infty} \frac{\langle u_i, g\rangle}{\sigma_i} v_i \right\|_2^2 = \sum_{i=1}^{\infty} \frac{|\langle u_i, g\rangle|^2}{\sigma_i^2} < \infty,$$

where the last inequality is called the Picard criterion. We will meet the Picard criterion in a discrete form later.

We will now demonstrate that the pseudo-inverse of K is unbounded. We disturb the right-hand side with a small perturbation $\hat{g} = g + \delta u_k$ in the kth component of the right-hand side. That is, $\|\hat{g} - g\|_2 = \delta$ for any choice of k. Comparing the solutions obtained by inserting the disturbed and the undisturbed right-hand side into (2.5) gives

$$\begin{aligned}\|K^\dagger g - K^\dagger(g+\delta u_k)\|^2 &= \left\| \sum_{i=1}^{\infty} \frac{\langle u_i, g\rangle}{\sigma_i} - \sum_{i=1}^{\infty} \frac{\langle u_i, g+\delta u_k\rangle}{\sigma_i} \right\|^2 \\ &= \frac{\delta^2 \langle u_k, u_k\rangle^2}{\sigma_k^2} = \frac{\delta^2}{\sigma_k^2}.\end{aligned}$$

Thus, if we let the index k of the disturbed component go toward infinity we get

$$\|K^\dagger g - K^\dagger(g+u_k)\|^2 \to \infty \text{ for } k \to \infty,$$

because δ is constant and $\sigma_i \to 0$ for $i \to \infty$. But $\|g - (g+\delta u_k)\|^2 = \delta^2$ is constant and we conclude that the pseudo-inverse is unbounded and even small changes to the right-hand side can completely and utterly destroy the solution. In fact, the disturbance δ does not need to be constant for all k for problems to appear. If the ratio $\delta_k^2/\sigma_k^2 \to \infty$ for $k \to \infty$ we end up with an unbounded solution. That is, we have an ill-posed problem.

Regularization

We can save some of the lost ground by applying *regularization* techniques which introduce a modified operator so that the inverse is bounded. The art is to modify the unbounded operator just right to obtain a solution as close to the desired result as possible with the given data.

The truncated SVE stops the summation in (2.5) at a truncation parameter k,

$$f_{\text{tsve}} = K_k^\dagger g = \sum_{i=0}^{k} \frac{\langle u_i, g\rangle}{\sigma_i} v_i.$$

We see that the solution is projected into the subspace spanned by $\{v_1, \ldots, v_k\}$ and the operator is obviously bounded as the sum is now finite. We will call a method like this a *projection* method or a *subspace* method.

Another approach is to penalize growth of the solution by adding an extra term to the minimization

$$f_{\text{tikh}} = K_\lambda^\dagger g = \underset{f}{\operatorname{argmin}}\{\|Kf-g\|_2^2 + \lambda^2\|f\|_2^2\}, \tag{2.6}$$

where the solution is bounded if the regularization parameter $\lambda > 0$. This approach is usually called Tikhonov regularization but we will also call it a *penalty* method due to the penalty term $\lambda^2\|f\|_2^2$.

After this first introduction to ill-posed problems in the continuous setting we turn to the tools used for the discretized problem. For a discussion of the discretization itself see for example [4, 30].

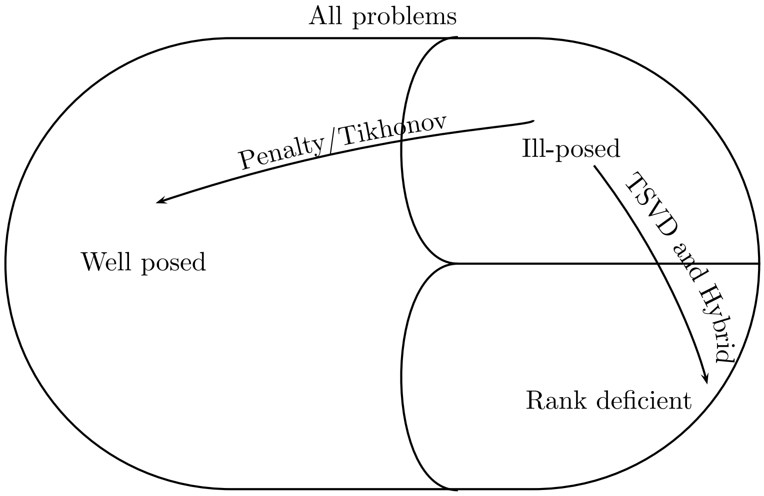


Figure 2.1: Types of regularization methods. The Tikhonov/penalty type methods create well-posed problems while the projection/subspace methods create rank-deficient problems. On this background it is somewhat surprising that truncated SVE and Tikhonov regularization can yield nearly identical results, see [64].

2.2 Matrix Decompositions

To analyze and solve the discrete ill-posed problems we use a number of matrix decompositions. The most important decomposition used to analyze discrete ill-posed problems is the singular value decomposition, the discrete analogue to the singular value expansion. The discrete Picard condition is also introduced and an example of the problems with a disturbed right hand side is presented.

Definition 2.1 (SVD) The singular value decomposition (SVD) of a matrix $\mathbf{K} \in \mathbb{R}^{m\times n}$, (assuming $m \geq n$) is

$$\mathbf{K} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T, \tag{2.7}$$

where $\mathbf{U} \in \mathbb{R}^{m\times m}$, $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$. Furthermore both matrices of singular vectors $\mathbf{U}$ (left singular vectors) and $\mathbf{V}$ (right singular vectors) are orthogonal with orthonormal columns, that is, $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^T\mathbf{V} = \mathbf{I}$. The singular value matrix

$\mathbf{\Sigma}$ is a "diagonal" matrix, i.e.,

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

with non-negative non-increasing sorted values, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$ on the main diagonal. If $m < n$ we have an SVD of the above form for $\mathbf{K}^T = \mathbf{U\Sigma V}^T$. Transposing gives $\mathbf{K} = \mathbf{V\Sigma}^T\mathbf{U}^T$, that is, $\mathbf{U}$ and $\mathbf{V}$ are swapped and $\mathbf{\Sigma}$ is transposed.

Definition 2.2 (Thin SVD) Let $p = \min(m, n)$ and define $\mathbf{U}_p = [\mathbf{u}_1 \ldots \mathbf{u}_p] \in \mathbb{R}^{m\times p}$ and $\mathbf{V}_p = [\mathbf{v}_1 \ldots \mathbf{v}_p] \in \mathbb{R}^{n\times p}$ as the matrices composed of the first p columns of $\mathbf{U}$ and $\mathbf{V}$ respectively. Letting $\mathbf{\Sigma}_p = \mathrm{diag}([\sigma_1 \ldots \sigma_p])$ be the $p \times p$ upper square part of $\mathbf{\Sigma}$, we define the *thin* SVD as

$$\mathbf{K} = \mathbf{U}_p\mathbf{\Sigma}_p\mathbf{V}_p^T.$$

The columns omitted in the thin SVD definition, i.e., $\mathbf{U}_0 = [\mathbf{u}_{p+1} \ldots \mathbf{u}_m]$ and $\mathbf{V}_0 = [\mathbf{v}_{p+1} \ldots \mathbf{v}_n]$ can be used to form the discrete analogues of $Q_u = \mathbf{U}_0\mathbf{U}_0^T$ and $Q_v = \mathbf{V}_0\mathbf{V}_0^T$ seen in (2.3) and (2.4).

The SVD reveals the rank of the operator as the largest index i with a singular value $\sigma_i > 0$. If $\mathrm{rank}(\mathbf{K}) < \min(m, n)$ we call $\mathbf{K}$ rank deficient as it has an null-space spanned by $[\mathbf{v}_{i+1} \ldots \mathbf{v}_n]$.

The SVD is related to the eigenvalue decomposition through the relations $\mathbf{KK}^T = \mathbf{U\Sigma}^2\mathbf{U}^T$ and $\mathbf{K}^T\mathbf{K} = \mathbf{V\Sigma}^2\mathbf{V}^T$, i.e., the non-zero singular values squared are the non-zero eigenvalues of the symmetric and positive semi-definite matrices $\mathbf{K}^T\mathbf{K}$ and $\mathbf{KK}^T$. Furthermore $\mathbf{U}$ and $\mathbf{V}$ form the corresponding eigenvectors.

The pseudo-inverse or Moore-Penrose generalized inverse $\mathbf{K}^\dagger$ already introduced for the continuous problem is easily converted to the discrete case using the SVD

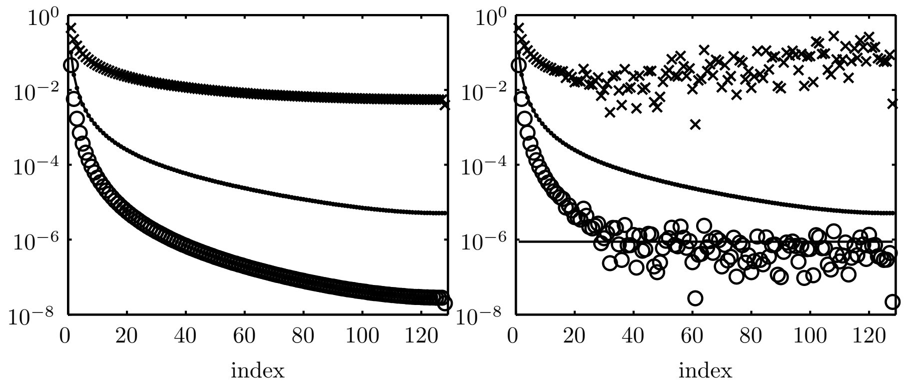
$$\mathbf{K}^\dagger\mathbf{y} = \sum_{i=1}^{\mathrm{rank}(\mathbf{K})} \frac{\mathbf{u}_i^T\mathbf{y}}{\sigma_i}\mathbf{v}_i \quad \text{that is} \quad \mathbf{K}^\dagger = \sum_{i=1}^{\mathrm{rank}(\mathbf{K})} \mathbf{v}_i\sigma_i^{-1}\mathbf{u}_i^T \tag{2.8}$$

Recall that the pseudo-inverse applied to the right-hand side yields the least squares solution of minimum norm. Because we have a finite sum the problem is, in the sense of Hadamard, in fact well-posed as the ill-posed problem has been regularized by the projection into the discrete problem. However, if the discretization is too fine (and we need such a fine discretization) the problems are unfortunately not over yet.

From the description of the pseudo-inverse we can now illustrate the problem by finding the least squares solution of minimum norm $\mathbf{K}^\dagger\mathbf{y}$. The singular values of an discrete ill-posed operator decrease fast and cluster near zero. For a solution to be *meaningful* (it always exists in the discrete case) $\mathbf{u}_i^T\mathbf{y}$ should decrease on average at least as fast as the singular values. Due to the resemblance with the Picard criterion this is called the discrete Picard condition [64, 66]. Similarly to the example shown in the previous section we also make an example with a disturbed right-hand side. Let the right-hand side be perturbed with a vector $\delta\mathbf{e}$, where $\mathbf{e}$ is a vector of white noise with norm $\|\mathbf{e}\|_2 = 1$. The white noise has components in all directions with equal probability, that is, the expected value $E(|\delta\mathbf{u}_i^T\mathbf{e}|^2) = \delta^2$. If we look at the solution with the disturbed right-hand side

$$\mathbf{K}^\dagger(\mathbf{y}+\mathbf{e}) = \sum_{i=1}^{p} \frac{\mathbf{u}_i^T\mathbf{y} + \mathbf{u}_i^T\mathbf{e}}{\sigma_i}\mathbf{v}_i,$$

we see that if $|\mathbf{u}_i^T\mathbf{y}| < \delta$ then the noise term is likely to dominate and the solution coefficient to $\mathbf{v}_i$ can turn out wrong. Due to the Picard criterion this is much more likely for large indices as $(\mathbf{u}_i^T\mathbf{y})^2$ should on average decrease at least as fast as σ_i^2. Figure 2.2 shows an example of how the coefficients behave with and without white noise added. With noise we see that $|\mathbf{u}_i(\mathbf{y} + \delta\mathbf{e})|$ levels off around δ instead of decreasing steadily. In this case we added the noise on purpose, but even the errors introduced by representing the right-hand side as floating point numbers can cause problems.



(a) Without noise (b) With noise

Figure 2.2: The dots show the singular values of an 128×128 DERIV2 problem. The circles ($\circ$) show the coefficients $|\mathbf{u}_i^T\mathbf{y}|$, while crosses ($\times$) show solution coefficients $|\mathbf{u}_i^T\mathbf{y}/\sigma_i|$. The noise dominates the small coefficients and the corresponding solution coefficients are too large because of small corresponding singular values. The straight line in (b) shows δ.

Just as the eigenvalue decomposition is related to the SVD the generalized eigenvalues have a cousin in the generalized SVD. Other definitions exist, see e.g. [46].

Definition 2.3 (GSVD) The generalized singular value decomposition (GSVD) of a pair $\mathbf{K} \in \mathbb{R}^{m\times n}$ and $\mathbf{L} \in \mathbb{R}^{p\times n}$ of matrices with the same number of columns and trivially intersecting null-spaces, i.e., $\mathcal{N}(\mathbf{K}) \cap \mathcal{N}(\mathbf{L}) = \{\mathbf{0}\}$, is defined

$$\begin{bmatrix} \mathbf{K} \\ \mathbf{L} \end{bmatrix} = \begin{bmatrix} \mathbf{U} & \mathbf{0} \\ \mathbf{0} & \mathbf{V} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Sigma} \\ \mathbf{M} \end{bmatrix} \mathbf{X}^{-1}, \tag{2.9}$$

where $\mathbf{U} \in \mathbb{R}^{m\times m}$, $\mathbf{V} \in \mathbb{R}^{p\times p}$, $\boldsymbol{\Sigma} \in \mathbb{R}^{m\times n}$, $\mathbf{M} \in \mathbb{R}^{p\times n}$, and $\mathbf{X} \in \mathbb{R}^{n\times n}$. Here $\mathbf{U}$ and $\mathbf{V}$ are orthogonal with orthonormal columns. The "diagonal" matrix $\boldsymbol{\Sigma}$ has non-negative elements in *non-decreasing* order along the main diagonal. The matrix $\mathbf{M}$ is "diagonal" and has non-negative elements in non-increasing order along the main diagonal. The values of $\boldsymbol{\Sigma}$ and $\mathbf{M}$ are scaled so that $\boldsymbol{\Sigma}^T\boldsymbol{\Sigma} + \mathbf{M}^T\mathbf{M} = \mathbf{I}$. Note that $\mathbf{X}$ is *not* orthogonal, only non-singular.

Any zero elements in the diagonal of $\boldsymbol{\Sigma}$, say the first k elements, are associated with the null-space of $\mathbf{K}$ as is the space spanned by the first k columns of $\mathbf{X}^{-1}$. The null-space of $\mathbf{L}$ is spanned by colums in $\mathbf{X}^{-1}$ where the associated diagonal element of $\mathbf{M}$ is zero (found in the end) and, if $p < n$, by the last $n - p$ columns of $\mathbf{X}^{-1}$.

2.3 Krylov Subspace Methods

The Lanczos tridiagonalization, Arnoldi "upper Hessenberg" and Lanczos bidiagonalization algorithms presented in the following form the basis of many iterative methods for symmetric, square and general matrices respectively. We will list the algorithms from the least generally applicable to the most applicable. However, we will first define the Krylov subspaces central to all three algorithms.

Definition 2.4 (Krylov Subspace) The Krylov subspace of dimension k generated by the square matrix $\mathbf{A}$ and vector $\mathbf{b}$ is defined

$$\mathcal{K}_k(\mathbf{A}, \mathbf{b}) = \operatorname{span}\{\mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}^2\mathbf{b}, \ldots, \mathbf{A}^{k-1}\mathbf{b}\}. \tag{2.10}$$

Lanczos Tridiagonalization

The Lanczos tridiagonalization algorithm listed in Alg. 1 is applicable to square and symmetric matrices and computes entries for a tridiagonal matrix as well as an orthonormal basis for a Krylov space. After k iterations with matrix $\mathbf{K}$ and starting vector $\mathbf{y}$ we have the following relation

$$\mathbf{K}\mathbf{V}_k = \mathbf{V}_k\mathbf{T}_k + \beta_k\mathbf{v}_{k+1}\mathbf{e}_k^T, \tag{2.11}$$

where $\mathbf{e}_k^T = [0 \ldots 0\, 1] \in \mathbb{R}^k$, and the columns of $\mathbf{V}_k = [\mathbf{v}_1 \ldots \mathbf{v}_k]$ form an orthonormal basis of the Krylov space $\mathcal{K}_k(\mathbf{K}, \mathbf{y})$, and $\mathbf{T}_k \in \mathbb{R}^{k\times k}$ is a symmetric tridiagonal matrix

$$\mathbf{T}_k = \begin{bmatrix} \alpha_1 & \beta_1 & & \cdots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_{k-1} \\ 0 & \cdots & & \beta_{k-1} & \alpha_k \end{bmatrix}. \tag{2.12}$$

If the algorithm is stopped due to $\beta_k = 0$ the Krylov space $\mathcal{K}_k(\mathbf{K}, \mathbf{y})$ defines an invariant subspace of $\mathbf{K}$.

The columns of $\mathbf{V}_k$ are orthonormal in exact arithmetic but cancellation in finite precision calculations destroys orthogonality. Infinite precision calculations can be simulated by adding a reorthogonalization step (either complete or selective) at the expense of extra work and extra storage [46, Sec. 9.2].

Algorithm 1: Lanczos Tridiagonalization.

$[\mathbf{V},\mathbf{T}] = \text{LanczosTri}\ (\mathbf{K},\mathbf{y},k)$

Performs a maximum of k steps of Lanczos tridiagonalization of $\mathbf{K}$ with $\mathbf{y}$ as starting vector. The matrix $\mathbf{V}$ forms an orthonormal basis of the Krylov subspace $\mathcal{K}_k(\mathbf{K}, \mathbf{y})$.

$\mathbf{v}_0 = \mathbf{0}$
$\beta_0 = \|\mathbf{y}\|_2; \quad \mathbf{r}_0 = \mathbf{y}$
$i \leftarrow 0$
while $\beta_i \neq 0$ and $i \leq k$ **do**
  $\mathbf{v}_{i+1} = \mathbf{r}_i / \beta_i$
  $\alpha_{i+1} = \mathbf{v}_{i+1}^T \mathbf{K} \mathbf{v}_{i+1}$
  $\mathbf{r}_{i+1} = \mathbf{K}\mathbf{v}_{i+1} - \alpha_{i+1}\mathbf{v}_{i+1} - \beta_i \mathbf{v}_i$
  $\beta_{i+1} = \|\mathbf{r}_{i+1}\|_2$
  $i \leftarrow i + 1$
end while
$\mathbf{V} = [\mathbf{v}_1 \ldots \mathbf{v}_i]$
Create $\mathbf{T}$ according to (2.12).

Arnoldi Upper Hessenberg

The Lanczos tridiagonalization process can only be applied to square and symmetric matrices. In the case of square *non-symmetric* matrices we have the unsymmetric Lanczos tridiagonalization algorithm and the Arnoldi method to choose from. We will only present Arnoldi because of the non-orthogonal (by construction) Lanczos vectors and breakdown problems seen in the unsymmetric Lanczos method, see e.g. [46].

The Arnoldi method creates an upper Hessenberg matrix instead of a symmetric tridiagonal matrix. After k steps of Alg. 2 we have the following relation

$$\mathbf{K}\mathbf{V}_k = \mathbf{V}_k\mathbf{H}_k + h_{k+1,k}\mathbf{v}_{k+1}\mathbf{e}_k^T$$

where $\mathbf{H}_k \in \mathbb{R}^{k\times k}$ is an upper Hessenberg matrix

$$\mathbf{H}_k = \begin{bmatrix} h_{11} & h_{12} & \cdots & \cdots & h_{1k} \\ h_{21} & h_{22} & \cdots & \cdots & h_{2k} \\ 0 & h_{32} & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{k,k-1} & h_{kk} \end{bmatrix}. \tag{2.13}$$

Similarly to the Lanczos tridiagonalization process the columns of $\mathbf{V}_k$ span the Krylov subspace $\mathcal{K}_k(\mathbf{K}, \mathbf{y})$ and if we get $h_{k+1,k} = 0$ then $\mathbf{V}_k$ defines an invariant subspace of $\mathbf{K}$. If we apply the Arnoldi process to a symmetric operator the result equals that of the Lanczos tridiagonalization, because the columns of $\mathbf{V}_k$ from the Arnoldi algorithm and $\mathbf{V}_k$ from Lanczos tridiagonalization are the same. Therefore, for a symmetric $\mathbf{K}$ we have that $h_{i,j} = 0$ for $i < j-1$ and $h_{i,j} = h_{j,i}$ if $i = j-1$. However, the entries are computed at the expense of more work during the orthogonalization process in each iteration. Due to the reorthogonalization step Arnoldi essentially simulates infinite precision computations.

Lanczos Bidiagonalizations

In case of a rectangular $\mathbf{K}$ we could use Lanczos tridiagonalization on the symmetric system $\mathbf{K}^T\mathbf{K}$ with $\mathbf{K}^T\mathbf{y}$ as starting vector or the symmetric system $\mathbf{K}\mathbf{K}^T$ with $\mathbf{y}$ as the starting vector. Another option is to use the Lanczos bidiagonalization algorithm (often attributed to Golub and Kahan [45]) that improves numerical stability by avoiding the explicit use of the $\mathbf{K}^T\mathbf{K}$ and $\mathbf{K}\mathbf{K}^T$.

Algorithm 3 creates from a matrix $\mathbf{K}$ and starting vector $\mathbf{y}$ a lower bidiagonal matrix and two orthogonal matrices with the relations

$$\mathbf{U}_{k+1}(\beta_0\mathbf{e}_1) = \mathbf{y} \tag{2.14}$$

$$\mathbf{K}\mathbf{V}_k = \mathbf{U}_{k+1}\mathbf{B}_k, \tag{2.15}$$

$$\mathbf{K}^T\mathbf{U}_{k+1} = \mathbf{V}_k\mathbf{B}_k^T + \alpha_{k+1}\mathbf{v}_{k+1}\mathbf{e}_{k+1}^T \tag{2.16}$$

where the lower bidiagonal matrix $\mathbf{B}_k \in \mathbb{R}^{k+1\times k}$ is created by

$$\mathbf{B}_k = \begin{bmatrix} \alpha_1 & & & \\ \beta_1 & \alpha_2 & & \\ & \beta_2 & \ddots & \\ & & \ddots & \alpha_k \\ & & & \beta_k \end{bmatrix}. \tag{2.17}$$

Algorithm 2: Arnoldi process

$$[\mathbf{V},\ \mathbf{H}] = \textsc{Arnoldi}(\mathbf{K},\mathbf{y},k)$$

Performs a maximum of k steps of the Arnoldi process on $\mathbf{K}$ with $\mathbf{y}$ as starting vector. The matrix $\mathbf{V}$ forms an orthonormal basis of the Krylov subspace $\mathcal{K}_k(\mathbf{K}, \mathbf{y})$.

$h_{1,0} = \|\mathbf{y}\|_2; \quad \mathbf{r}_0 = \mathbf{y}$
$i \leftarrow 0$
while $i \leq k$ and $h_{i+1,i} \neq 0$ **do**
    $\mathbf{v}_{i+1} = \mathbf{r}_i / h_{i+1,i}$
    $i \leftarrow i + 1$
    $\mathbf{r}_i \leftarrow \mathbf{K}\mathbf{v}_i$
    for $j = 1 : i$ **do**
        $h_{j,i} = \mathbf{r}_i^T \mathbf{v}_j$
        $\mathbf{r}_i \leftarrow \mathbf{r}_i - h_{j,i}\mathbf{v}_j$
    end for
    $h_{i+1,i} = \|\mathbf{r}_i\|_2$
end while
$\mathbf{V} = [\mathbf{v}_1 \dots \mathbf{v}_i]$
Create $\mathbf{H}$ according to (2.13).

The columns of $\mathbf{V}_k$ span the Krylov space $\mathcal{K}_k\left(\mathbf{K}^T\mathbf{K}, \mathbf{K}^T\mathbf{y}\right)$ while the columns of $\mathbf{U}_{k+1}$ span the Krylov space $\mathcal{K}_{k+1}\left(\mathbf{K}\mathbf{K}^T, \mathbf{y}\right)$. The columns will be orthogonal if exact arithmetic is used, i.e., $\mathbf{V}_k^T\mathbf{V}_k = \mathbf{I}$ and $\mathbf{U}_{k+1}^T\mathbf{U}_{k+1} = \mathbf{I}$. A reorthogonalization step can be added after the computation of new vectors $\mathbf{v}_i$ and $\mathbf{u}_{i+1}$ to simulate infinite precision calculations.

The lower bidiagonalization of $\mathbf{K}$ with $\mathbf{y}$ as starting vector is related to Lanczos tridiagonalization of $\mathbf{K}^T\mathbf{K}$ with starting vector $\mathbf{K}^T\mathbf{y}$. They both compute the same $\mathbf{V}_k$ because they both span $\mathcal{K}_k\left(\mathbf{K}^T\mathbf{K}, \mathbf{K}^T\mathbf{y}\right)$. If we multiply (2.15) with $\mathbf{V}_k^T\mathbf{K}^T$ and insert (2.16) we get

$$\begin{aligned}
\mathbf{V}_k^T\mathbf{K}^T\mathbf{K}\mathbf{V}_k &= \mathbf{V}_k^T\mathbf{K}^T\mathbf{U}_{k+1}\mathbf{B}_k \\
&= \mathbf{V}_k^T\mathbf{V}_k\mathbf{B}_k^T\mathbf{B}_k + \alpha_{k+1}\mathbf{V}_k^T\mathbf{v}_{k+1}\mathbf{e}_{k+1}^T\mathbf{B}_k \\
&= \mathbf{B}_k^T\mathbf{B}_k.
\end{aligned}$$

Comparing with (2.11) we conclude that $\mathbf{T}_k = \mathbf{B}_k^T\mathbf{B}_k$. That is, we have a Cholesky-like factorization of the tridiagonal matrix $\mathbf{T}_k$ from the Lanczos tridiagonalization algorithm. If we instead multiply equation (2.16) with $\mathbf{U}_{k+1}^T\mathbf{K}$ and insert (2.15) we get

$$\begin{aligned}
\mathbf{U}_{k+1}^T\mathbf{K}\mathbf{K}^T\mathbf{U}_{k+1} &= \mathbf{U}_{k+1}^T\mathbf{K}\mathbf{V}\mathbf{B}_k^T + \alpha_{k+1}\mathbf{U}_{k+1}^T\mathbf{K}\mathbf{v}_{k+1}\mathbf{e}_{k+1} \\
&= \mathbf{U}_{k+1}^T\mathbf{U}_{k+1}\mathbf{B}_k\mathbf{B}_k^T + \alpha_{k+1}\mathbf{U}_{k+1}^T\mathbf{K}\mathbf{v}_{k+1}\mathbf{e}_{k+1}
\end{aligned}$$

and using lines 7–8 of the algorithm to substitute $\mathbf{K}\mathbf{v}_{k+1}$ we get

$$\begin{aligned}
\mathbf{U}_{k+1}^T\mathbf{K}\mathbf{K}^T\mathbf{U}_{k+1} &= \mathbf{U}_{k+1}^T\mathbf{U}_{k+1}\mathbf{B}_k\mathbf{B}_k^T + \alpha_{k+1}\mathbf{U}_{k+1}^T(\beta_{k+1}\mathbf{u}_{k+2} + \alpha_{k+1}\mathbf{u}_{k+1})\mathbf{e}_{k+1} \\
&= \mathbf{B}_k\mathbf{B}_k^T + \alpha_{k+1}^2\mathbf{e}_{k+1}\mathbf{e}_{k+1}^T.
\end{aligned}$$

The right-hand side must be equal to the tridiagonal matrix obtained from Lanczos tridiagonalization of $\mathbf{K}\mathbf{K}^T$ with $\mathbf{y}$ as starting vector because $\mathbf{U}_{k+1}$ spans the Krylov space $\mathcal{K}_k\left(\mathbf{K}\mathbf{K}^T, \mathbf{y}\right)$. That is, $\bar{\mathbf{T}}_{k+1} = \mathbf{B}_k\mathbf{B}_k^T + \alpha_{k+1}^2\mathbf{e}_{k+1}\mathbf{e}_{k+1}^T$ and we conclude that the leading $k \times k$ part of $\mathbf{B}_k$, that is,

$$\bar{\mathbf{B}}_k = \begin{bmatrix} \alpha_1 & & & \\ \beta_1 & \alpha_2 & & \\ & \beta_2 & \ddots & \\ & & \ddots & \alpha_k \end{bmatrix} \in \mathbb{R}^{k\times k}$$

is related to the tridiagonal matrix from Lanczos tridiagonalization via $\bar{\mathbf{T}}_k = \bar{\mathbf{B}}_k\bar{\mathbf{B}}_k^T$. These relations, linking Lanczos tridiagonalization and bidiagonalization, are used in §5.1.

The Krylov subspace methods have in their various forms led to many iterative solvers. Behind the LSQR method [107] lies the bidiagonalization to lower form, behind GMRES [116] lies the Arnoldi process, and Lanczos tridiagonalization forms the basis of MINRES [105]—just to mention a few. For an overview of these and other solvers see, for example, the "template" book [5].

Algorithm 3: Lanczos Bidiagonalization

$[\mathbf{U},\mathbf{V},\mathbf{B}] = \textsc{bidiag}(\mathbf{K},\mathbf{y},\ k)$

Performs a maximum of k steps of bidiagonalization on $\mathbf{K}$ with $\mathbf{y}$ as starting vector. The matrix $\mathbf{V}$ forms an orthogonal basis of the Krylov space $\mathcal{K}_k\left(\mathbf{K}^T\mathbf{K}\,,\mathbf{K}^T\mathbf{y}\right)$ while $\mathbf{U}$ forms an orthogonal basis of the Krylov space $\mathcal{K}_{k+1}\left(\mathbf{K}\mathbf{K}^T\,,\mathbf{y}\right)$.

$\beta_0 = \|\mathbf{y}\|_2; \quad \mathbf{u}_1 = \mathbf{y}/\beta_0$
$\mathbf{v}_0 = \mathbf{0}$
$i \leftarrow 0$
while $i \le k$ and $\beta_i > 0$ **do**
  $\mathbf{p}_{i+1} = \mathbf{K}^T\mathbf{u}_{i+1} - \beta_i\mathbf{v}_i$
  $i \leftarrow i+1$
  $\alpha_i = \|\mathbf{p}_i\|_2; \quad \mathbf{v}_i = \mathbf{p}_i/\alpha_i$
  $\mathbf{q}_i = \mathbf{K}\mathbf{v}_i - \alpha_i\mathbf{u}_i$
  $\beta_i = \|\mathbf{q}_i\|_2; \quad \mathbf{u}_{i+1} = \mathbf{q}_i/\beta_i$
end while
$\mathbf{U} = [\mathbf{u}_1 \dots \mathbf{u}_{i+1}]; \quad \mathbf{V} = [\mathbf{v}_1 \dots \mathbf{v}_i]$
Create $\mathbf{B}$ according to (2.17).

2.4 The Givens Transformation

The Givens rotation is useful for selectively zeroing elements and will be heavily used in connection with bidiagonal systems in §5.1.

An $n \times n$ Givens rotation is described by

$$\mathbf{G}_{ij}(\theta) = \begin{array}{c} \\ \\ \\ i \\ \\ j \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & i & & j & & \end{array} \\ \left[\begin{array}{ccccccc} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & & s & & \\ & & & \ddots & & & \\ & & -s & & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{array}\right] \end{array}, \tag{2.18}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$. It is called a rotation because multiplying a vector with $\mathbf{G}_{ij}(\theta)$ corresponds to rotating the vector in the (i,j) coordinate plane. Applying a Givens rotation to a vector $\mathbf{a} = [a_1, \dots, a_n]$, i.e., $\mathbf{b} = \mathbf{G}_{ij}\mathbf{a}$ only modifies element i and j of $\mathbf{a}$. That is,

$$b_k = \begin{cases} ca_i + sa_j & \text{if } k = i \\ -sa_i + ca_j & \text{if } k = j \\ a_k & \text{otherwise.} \end{cases}$$

Thus, to create a zero in element j we choose

$$c = a_i/\gamma, \quad s = a_j/\gamma, \quad \gamma = \sqrt{a_i^2 + a_j^2},$$

where $\gamma \neq 0$ unless we already have a 0 in both elements. Algorithm 4 computes c, s and γ while avoiding overflow.

Algorithm 4: Givens coefficients

$$[\gamma,\ c,\ s] = \text{GIVENS}(\ \alpha,\ \beta)$$

Compute γ, c and s for a Givens rotation such that β is zeroed, that is, $-s\alpha + c\beta = 0$ and avoid overflow. Adapted from [13].

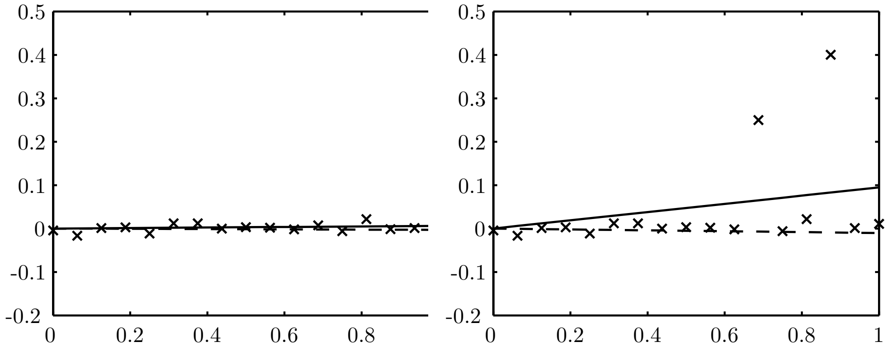
if $\beta = 0$ **then**
  $c = 1; \quad s = 0; \quad \gamma = \alpha$
else if $|\beta| > |\alpha|$ **then**
  $t = \alpha/\beta; \quad u = \sqrt{1 + t^2}$
  $s = 1/u; \quad c = ts; \quad \gamma = u\beta$
else
  $t = \beta/\alpha; \quad u = \sqrt{1 + t^2}$
  $c = 1/u; \quad s = tc; \quad \gamma = u\alpha$
end if

2.5 Robust Estimation

Our standard problem $\mathbf{Kx} = \mathbf{y}$ can also be seen in the context of a data-fitting or an estimation problem where we fit a linear combination of the columns of $\mathbf{K}$ to the data vector $\mathbf{y}$. The usual approach is to use the least squares solution

$$\mathbf{x} = \underset{\mathbf{x}}{\operatorname{argmin}} \|\mathbf{Kx} - \mathbf{y}\|_2^2.$$

The Gauss-Markov theorem states that if the errors have expectation zero, have equal variance and are uncorrelated then the least squares solution gives the best (that is, minimum variance) linear unbiased estimate of $\mathbf{x}$, see [13, Theorem 1.1.1] and [31]. However, the noise and errors in the right-hand side does not always “play” nice with the least squares approach. The least squares solution is excessively sensitive to outliers in $\mathbf{y}$, that is, elements with an error much larger than others. In Fig. 2.3(b) we see the least squares solution trying to approximate the two outliers at the expense of the approximation of all other points.



(a) Normal distributed noise

(b) Normal distributed noise and two outliers

Figure 2.3: Fitting of a straight line with least squares (solid) and least absolute sum (dashed) to two sets of data. The operator $\mathbf{K} = [0\, 1/16 \cdots 1]^T \in \mathbb{R}^{18\times 1}$ and the undisturbed right-hand side is $\mathbf{y} = \mathbf{0}$, with the obvious solution $\mathbf{x} = 0$. The random normal distributed noise is the same in both examples.

One obvious remedy is to somehow detect the outliers and remove them from the dataset—an easy task in our rather extreme example shown in Fig. 2.3(b). However, in automated processing of data or in cases with less pronounced outliers, other methods need to be considered. The "robust estimation" methods can handle outliers much better than least squares. The least sum of absolute values solution

$$\mathbf{x} = \operatorname*{argmin}_{\mathbf{x}} \|\mathbf{K}\mathbf{x} - \mathbf{y}\|_1,$$

is such a robust estimation method. The dashed line in Fig. 2.3(b) is less affected by the two outliers compared to the solutions obtained from a right-hand side without outliers, cf. Fig. 2.3(a). To explain this we use the following statistical considerations.

The least squares approach is a maximum likelihood method for noise with each element distributed after the probability density function

$$p_2(e) = \frac{1}{\sqrt{\pi}} \exp\left(e^2\right). \tag{2.19}$$

That is, the probability of an error $|e_i| > \delta$ is

$$P_2[|e_i| > \delta] = P_2[e_i < -\delta] + P_2[e_i > \delta] = \int_{-\infty}^{-\delta} p_2(t)dt + \int_{\delta}^{\infty} p_2(t)dt.$$

The least sum of absolute values is a maximum likelihood estimator for noise with elements distributed after the probability density function

$$p_1(e) = \frac{1}{2}\exp\left(|e|\right), \tag{2.20}$$

also called the double-exponential or Laplace distribution [31]. Figure 2.4 shows the probability densities (2.19) and (2.20). Notice that the "tails", i.e., the area under the graphs for errors larger than say ± 2.5, are almost zero for the p_2 probability density function but not so for the p_1 function. Due to the "shorter tail" the least squares solution does not accept large misfits for any points and reduces them by increasing the error off all other points.

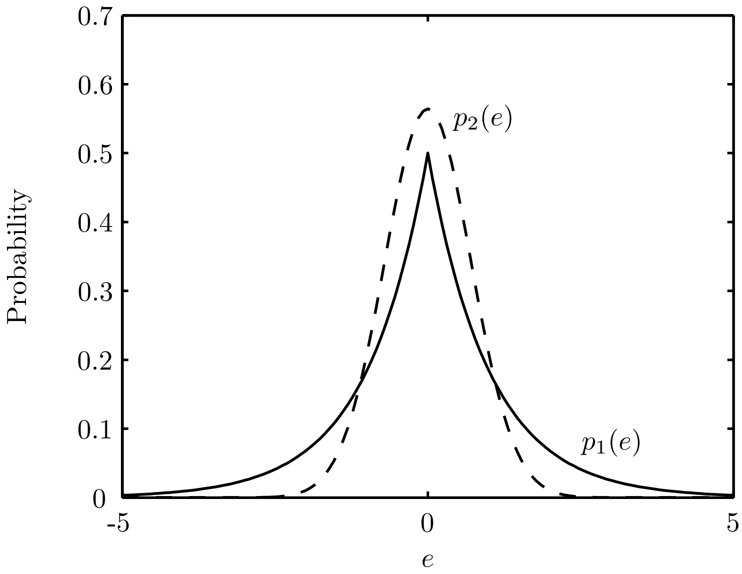


Figure 2.4: The normal probability density function $p_2(e) = \exp(e^2)/\sqrt{2}$ (dashed line) and the l_1-norm probability density function $p_1(e) = \exp(|x|)/2$ (solid line). The normal density function is more concentrated around 0 than the Laplace distribution function that instead has larger and longer tails.

We have seen that using other norms for measuring the residual can adapt to special kinds of errors. However, using another norm for penalty terms, such as $\|\mathbf{x}\|_2^2$, in (2.6) also has its uses. A penalty term $\|\mathbf{x}\|_2^2$ forces all elements toward zero. But using, for example, $\|\mathbf{x}\|_1$ allows for outliers in $\mathbf{x}$, that is, "discontinuities" in $\mathbf{x}$ which on a plot of $\mathbf{x}$ would appear as "spikes".

2.5.1 Robust Penalty Functions

In this section we list some of useful robust norm-like functions, that we will call penalty functions.

In the linear algebra society the most pronounced norm is probably the squared l_2-norm defined as

$$\|\mathbf{x}\|_2^2 = \sum x_i^2.$$

The popularity is due to a number of nice properties:

- The l_2-norm is invariant to orthonormal transformations, i.e., if $\mathbf{Q}$ is an square orthogonal matrix with orthonormal columns we have $\|\mathbf{x}\|_2^2 = \|\mathbf{Qx}\|_2^2$.
- The gradient of the l_2-norm squared exists and is simple

  $$\nabla\|\mathbf{x}\|_2^2 = 2\mathbf{x},$$

  and the Hessian is even more simple (and extremely easy to invert!)

  $$\mathbf{H}(\|\mathbf{x}\|_2^2) = 2\mathbf{I}.$$

- Minimizing the model fit in the two norm $\|\mathbf{Kx} - \mathbf{y}\|_2^2$ is statistically appropriate as the least squares solution is the unbiased linear estimator of minimum variance if the errors are uncorrelated, have zero means and equal variance.
- In 1, 2 and 3 dimensions the l_2-norm corresponds to our usual concept of distance.

However, if the errors in $\mathbf{y}$ are not normally distributed we have seen that the l_1-norm might be more useful. The l_1-norm is defined by

$$\|\mathbf{x}\|_1 = \sum |x_i|.$$

The l_2 and l_1-norms are members of the l_p-norm family, defined

$$\|\mathbf{x}\|_p = \left(\sum |x_i|^p\right)^{1/p}, \quad p \geq 1. \tag{2.21}$$

We will only consider l_p-norms with $p \leq 2$ because norms with $p > 2$ are not robust. Consider the minimization of the extreme l_∞-norm case where the maximum residual element is minimized (and therefore also called minimax optimization). This corresponds to an assumption of an uniform error distribution on $[-d, d]$ for some d. In this case there is absolutely no tail.

The gradient vector $\nabla\|\mathbf{x}\|_p^p$ of the l_p-norm to the pth power has the elements

$$[\nabla\|\mathbf{x}\|_p^p]_i = p|x_i|^{p-1}\text{sign}(x_i),$$

where the sign function is defined

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0. \end{cases}$$

The Hessian of $\|\mathbf{x}\|_p^p$ is (assuming $x_i \neq 0$) a diagonal matrix $\mathbf{H}(\|\mathbf{x}\|_\mathbf{p}^\mathbf{p})$ with the diagonal elements

$$[\mathbf{H}(\|\mathbf{x}\|_p^p)]_{i,i} = p(p-1)|x_i|^{p-2}.$$

We see that $|x_i|^{(p-2)}$ is not defined for any $x_i = 0$ if $p < 2$ and as a consequence neither is the Hessian. Thus we cannot use the efficient solvers that use second order information. Furthermore, if $p = 1$ we have an all-zero Hessian $\mathbf{H}(\|\mathbf{x}\|_1) = \mathbf{0}$!

To avoid the discontinuous gradients and all-zero Hessians we can use other more exotic norm-like functions (called M-type estimators by Huber [78]) defined as the sum

$$F(\mathbf{x}) = \sum_{i=1}^{n} f(x_i), \tag{2.22}$$

where the function f is a convex function, for example, chosen from Table 2.1. The gradient and Hessian are then

$$\nabla F(\mathbf{x}) = \begin{bmatrix} f'(x_1) \\ f'(x_2) \\ \vdots \\ f'(x_n) \end{bmatrix}, \mathbf{H}(F(\mathbf{x})) = \begin{bmatrix} f''(x_1) & & & \\ & f''(x_2) & & \\ & & \ddots & \\ & & & f''(x_n) \end{bmatrix}. \tag{2.23}$$

Which penalty function to use is not an easy choice. Furthermore they all introduce an extra parameter to adjust. The Logistic and Fair penalty functions have the advantage that they do not branch depending on the particular values of the argument. Ekblom [32] conducted experiments showing that the Huber function is a better choice than the l_1-norm. More information on l_p-norm estimation is found in [51] including methods to choose the optimal p.

With the gradient and Hessian available we are able to use these penalty functions in most standard optimization routines for finding robust solutions.

2.6 Optimization and Optimality

As we will see in the next chapter all our problems reduce to problems of the type

$$\min \sum_{i=1}^{q} \lambda_i F_i(\mathbf{B}_i\mathbf{x} - \mathbf{z}_i) = \min \sum_{i=1}^{q} \lambda_i F_i(\mathbf{r}_i), \tag{2.24}$$

Name		$f(z)$	$f'(z)$	$f''(z)$
l_p-norm	$\lVert\cdot\rVert_p^p$	$\lvert z\rvert^p$	$p\lvert z\rvert^{p-1}\operatorname{sign}(z)$	$p(p-1)\lvert z\rvert^{p-1}$
Logistic	f_L	$\beta^2\log(\cosh(z/\beta))$	$\beta\tanh(z/\beta)$	$1-\tanh^2(z/\beta)$
Fair	f_F	$\beta^2(\lvert z\rvert/\beta-\log(1+\lvert z\rvert/\beta))$	$\beta z/(\beta+\lvert z\rvert)$	$\beta^2/(\beta+\lvert z\rvert)^2$
Huber	f_H	$\begin{cases} z^2/2 \\ \beta\lvert z\rvert-\beta^2/2 \end{cases}$	$\begin{cases} z \\ \operatorname{sign}(z) \end{cases}$	$\begin{cases} 1 & \text{if } \lvert z\rvert<\beta \\ 0 & \text{if } \lvert z\rvert\geq\beta \end{cases}$

Table 2.1: Examples of penalty functions and their first and second derivatives. The scalar parameter β adjusts when point is considered an outlier. Collected from [6, 26, 103].

where $\mathbf{r}_i$ is the ith residual vector $\mathbf{r}_i = \mathbf{B}_i\mathbf{x} - \mathbf{z}_i$ and each F_i function is defined as a sum

$$F_i(\mathbf{r}_i) = \sum_j f_i(r_j),$$

where the function f_i is convex.

That is, we have a sum of composite functions each composed of a possible non-linear function and a linear function. To shorten notation we will use the notation

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \\ \vdots \\ \mathbf{B}_n \end{bmatrix}, \mathbf{z} = \begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \\ \vdots \\ \mathbf{z}_n \end{bmatrix} \text{ and } F(\mathbf{B}\mathbf{x}-\mathbf{z}) = \sum_{i=1}^{q} \lambda_i F_i(\mathbf{B}_i\mathbf{x}-\mathbf{z}_i).$$

In particular, the least squares standard-form Tikhonov problem has $q = 2$, $\mathbf{B}_1 = \mathbf{K}$, $\mathbf{z} = \mathbf{y}$, $\mathbf{B}_2 = \mathbf{I}$, $\mathbf{z}_2 = \mathbf{0}$, $F_1(\mathbf{r}) = \|\mathbf{r}\|_2^2$, $\lambda_1 = 1$, $F_2(\mathbf{r}) = \|\mathbf{r}\|_2^2$ and λ_2 is our usual regularization parameter. We will now derive gradients and Hessians for this special type of function and state optimality conditions for the problem in (2.24).

The chain rule gives us the gradient with respect to $\mathbf{x}$ from the gradients with respect to each residual

$$\nabla_{\mathbf{x}} F(\mathbf{B}\mathbf{x}-\mathbf{z}) = \mathbf{B}^T\nabla_{\mathbf{r}}F(\mathbf{r}) = \mathbf{B}^T \begin{bmatrix} \lambda_1\nabla_{\mathbf{r}_1}F_1(\mathbf{r}_1) \\ \lambda_2\nabla_{\mathbf{r}_2}F_2(\mathbf{r}_2) \\ \vdots \\ \lambda_q\nabla_{\mathbf{r}_q}F_q(\mathbf{r}_q) \end{bmatrix}.$$

If we create a block diagonal matrix of the diagonal Hessian matrices with respect to each residual

$$\mathbf{H}_{\mathbf{r}}(F(\mathbf{r})) = \begin{bmatrix} \mathbf{H}_{\mathbf{r}_1}(\lambda_1 F_1(\mathbf{r}_1)) & & & \\ & \mathbf{H}_{\mathbf{r}_2}(\lambda_2 F_2(\mathbf{r}_2)) & & \\ & & \ddots & \\ & & & \mathbf{H}_{\mathbf{r}_q}(\lambda_q F_n(\mathbf{r}_q)) \end{bmatrix},$$

we find by the chain rule the Hessian with respect to $\mathbf{x}$

$$\mathbf{H}_{\mathbf{x}}(F(\mathbf{x})) = \mathbf{B}^T \mathbf{H}_{\mathbf{r}}(F(\mathbf{r}))\mathbf{B}.$$

Because we will only consider convex functions $f_i(\mathbf{r})$ a sufficient condition for the solution is simply

$$\nabla F(\mathbf{r}) = \mathbf{0}. \tag{2.25}$$

If we return to the least squares standard-form Tikhonov example we have the optimality condition

$$\begin{bmatrix} \mathbf{K} \\ \lambda^2 \mathbf{I} \end{bmatrix}^T \begin{bmatrix} \mathbf{K}\mathbf{x} - \mathbf{y} \\ \mathbf{x} \end{bmatrix} = (\mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{I})\mathbf{x} - \mathbf{K}^T\mathbf{y} = 0.$$

Linear Equality Constraints

If we add linear equality constraints to the generic problem, we have a problem of the type

$$\min F(\mathbf{B}\mathbf{x} - \mathbf{z}) \quad \text{subject to} \quad \mathbf{C}\mathbf{x} = \mathbf{d}, \tag{2.26}$$

we need to extend the simple condition for optimality (2.25) to somehow include the constraints. The *Lagrangian* function for the problem (2.26) is

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = F(\mathbf{B}\mathbf{x} - \mathbf{z}) - \boldsymbol{\lambda}^T(\mathbf{C}\mathbf{x} - \mathbf{d}), \tag{2.27}$$

where $\boldsymbol{\lambda}$ is a vector of so-called Lagrange multipliers. The Lagrangian is used to formulate the first-order necessary conditions (the Karush-Kuhn-Tucker (KKT) conditions) for a solution

$$\begin{aligned} \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = \mathbf{B}^T \nabla F(\mathbf{r}) - \mathbf{C}^T \boldsymbol{\lambda} &= \mathbf{0} \\ \mathbf{C}\mathbf{x} &= \mathbf{d}, \end{aligned}$$

see [39, Theorem 9.1.1] for proof and a standard reference on optimization. In our case of a convex function F and linear equality constraints (that are convex) the KKT-conditions are also sufficient conditions for a solution.

Non-Negativity Constraints

Problems with non-negativity constraints are not as simple as problems with equality constraints. A non-negativity constraint $x_i \geq 0$ is called active if $x_i = 0$ or inactive if $x_i > 0$. The active set is the set of active constraints which we will denote $\mathcal{A}(\mathbf{x}) = \{i|x_i = 0\}$. Note that if the active set at the solution is known we effectively have a problem with linear equality constraints. The active set methods, see [39], try to find the active constraints at the solution and thereby reducing the problem to the simpler equality constrained problem.

The KKT conditions for a solution of a non-negativity constrained problem

$$\min F(\mathbf{B}\mathbf{x} - \mathbf{z}), \text{ subject to } \mathbf{x} \geq \mathbf{0}$$

are

$$\begin{aligned} \nabla_{\mathbf{x}} F(\mathbf{B}\mathbf{x} - \mathbf{z}) - \boldsymbol{\lambda} &= \mathbf{0} \\ \mathbf{x} &\geq \mathbf{0} \\ \boldsymbol{\lambda} &\geq \mathbf{0} \\ \mathbf{x} \odot \boldsymbol{\lambda} &= \mathbf{0}, \end{aligned}$$

where $\odot$ is the elementwise multiplication. In our special case we can use the first equation to eliminate $\boldsymbol{\lambda}$:

$$\begin{aligned} \nabla_{\mathbf{x}} F(\mathbf{B}\mathbf{x} - \mathbf{z}) &\geq \mathbf{0} \\ \mathbf{x} &\geq \mathbf{0} \\ \mathbf{x} \odot \nabla_{\mathbf{x}} F(\mathbf{B}\mathbf{x} - \mathbf{z}) &= \mathbf{0}. \end{aligned}$$

Again we refer to [39] for a more in depth treatment of optimization. For this particular scenario see also [124, Chap. 9]. Because we have restricted ourselves to linear constraints the constraint qualifications are fulfilled.

CHAPTER 3

Modular Algorithms

The results indicate that the method is very suitable for high speed machines. Hestenes and Stiefel, "Methods of Conjugate Gradients for solving Linear Systems" [75].

... when a traveler reaches a fork in the road, the L_1 norm tells him to take either way or the other, but the L_2 norm instructs him to head off into the bushes. Claerbout and Muir, "Robust Modeling with Erratic Data" [25]

In this chapter we take a look at existing regularization methods to determine if it is possible to either

1. split them into independent modules, that can be combined,
2. or form a common framework, that can handle a class of problems by changing parameters.

Then we will demonstrate how it is possible to combine the modules and exploit the common framework to form new regularization methods with other properties than those we started out with.

3.1 Two Regularization Classes

Probably the two most important regularization methods are:

Tikhonov regularization. A *penalty* term is added to the problem to filter out unwanted components.

Truncated SVD regularization (TSVD). The solution is projected into a specific *subspace* without the unwanted components.

These two methods will form the basis of a splitting of the regularization methods into two classes called penalty and projection methods, respectively. Note that it is not a clear-cut splitting as many methods can be formulated so that they appear in both groups. However, the formulation of a method is often more natural for one class than the other. Finally, a projection method can be combined with a penalty method (or another projection method) to form a so-called *hybrid* method. The following pages go more into depth with the different regularization approaches.

3.1.1 Penalty Methods

This section explains the class of penalty methods. In § 3.2.1 we go into more details about algorithms.

The usual standard-form Tikhonov regularization setup for a linear ill-posed problem $\mathbf{Kx} = \mathbf{y}$ is

$$\mathbf{x}_\lambda = \underset{\mathbf{x}}{\operatorname{argmin}} \left\{ \|\mathbf{Kx} - \mathbf{y}\|_2^2 + \lambda^2 \|\mathbf{x}\|_2^2 \right\}, \tag{3.1}$$

where we denote $\|\mathbf{Kx} - \mathbf{y}\|_2^2$ the model fit and $\|\mathbf{x}\|_2^2$ the penalty term. We will in the following look at variations of (3.1). How to solve the problem is the topic of §3.2.1. The choice of regularization parameter λ is also important and §3.3 is used for a discussion of parameter choice methods.

The Tikhonov formulation (3.1) has several possibilities for generalizations. One is to modify the penalty term to $\|\mathbf{Lx}\|_2^2$, where $\mathbf{L}$ is a matrix describing, for example, the first derivative. Knowledge of the covariance matrix $\mathbf{CC}^T$ of the solution vector $\mathbf{x}$ can also be used through the penalty term $\|\mathbf{C}^{-1}\mathbf{x}\|$, see [122].

Other possibilities are to use more than one penalty term and change the type of norm of the model fit and the penalty term(s). Furthermore actual constraints, such as non-negativity of the solution, can be added depending on the physics of the particular problem.

That is, we consider a general Tikhonov regularization formulation of the form

$$\mathbf{x}_{\text{reg}} = \underset{\mathbf{x}}{\operatorname{argmin}} \left\{ F_0(\mathbf{Kx} - \mathbf{y}) + \sum_{i=1}^{q} \lambda_i F_i(\mathbf{L}_i(\mathbf{x} - \mathbf{x}^*)) \right\}, \tag{3.2}$$

where $\mathbf{x}^*$ is an a priori guess of the solution, q is the number of penalty terms, $\mathbf{L}_i$ are regularization matrices, λ_i are corresponding regularization parameters, and F_i are "penalty" functions, see §2.5.1. The classical setup is $F_0(\mathbf{r}) = \|\mathbf{r}\|_2^2$, $F_1(\mathbf{r}) = \|\mathbf{r}\|_2^2$, $q = 1$ and no a priori guess, that is, $\mathbf{x}^* = \mathbf{0}$. The generalization can be extended to include constraints such as non-negativity, linear constraints $\mathbf{Gx} = \mathbf{0}$, etc. Several already proposed regularization setups can be found as combinations of the functions listed in Table 3.1.

From (3.2) we realize that a solution method cannot be divided into small modules, each minimizing one part of the problem, because changing the value of $\mathbf{x}$ influences both the model fit $F_0(\mathbf{Kx} - \mathbf{y})$ as well as the penalty terms $\lambda_i F_i(\mathbf{L}_i(\mathbf{x} - \mathbf{x}^*))$. Instead we will develop a general framework for the solution of (3.2), that uses the penalty functions provided as parameters.

$F_0(\mathbf{r})$	$F_i(\mathbf{r})$	Constraints
$\|\mathbf{r}\|_2^2$	$\|\mathbf{r}\|_2^2$	$\mathbf{x} \geq \mathbf{0}$
$\|\mathbf{r}\|_q^q$	$\|\mathbf{r}\|_p^p$	$\mathbf{Gx} \geq \mathbf{0}$
$\|\mathbf{r}\|_1$	$\|\mathbf{r}\|_1$	$\mathbf{Gx} = \mathbf{0}$
$\vdots$	$\sum x_i \log(w_i x_i)$	$\|\mathbf{x}\|_s = \delta$
	$\vdots$	$\vdots$

Table 3.1: Examples of norms, penalty functions and constraints that can be used with Tikhonov regularization. To ensure convex functions $1 < p, q$. See also Table 2.1. for more examples of $F_0(\mathbf{r})$ and $F_i(\mathbf{r})$.

Penalty or Constraints

The distinction between constraints and penalty terms is somewhat arbitrary. For example, the constrained problem

$$\min \|\mathbf{Kx} - \mathbf{y}\|_2^2 \text{ subject to } \|\mathbf{x}\|_2^2 = \delta,$$

has the Lagrangian function (see (2.27))

$$\mathcal{L}(\mathbf{x}, \lambda) = \|\mathbf{Kx} - \mathbf{y}\|_2^2 - \lambda(\delta - \|\mathbf{x}\|_2^2).$$

The stationary point $(\mathbf{x}_*, \lambda_*)$ of the Lagrangian, that is, the point $(\mathbf{x}, \lambda)$ fulfilling

$$\nabla\mathcal{L}(\mathbf{x}, \lambda) = \begin{bmatrix} \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}, \lambda) \\ \nabla_{\lambda}\mathcal{L}(\mathbf{x}, \lambda) \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix},$$

solves the constrained problem because both the objective and constraint functions are convex. If we assume that the stationary λ_* has been found we have that the stationary point of

$$\mathcal{L}(\mathbf{x}) = \|\mathbf{Kx} - \mathbf{y}\|_2^2 + \lambda_*(\delta - \|\mathbf{x}\|_2^2),$$

is also the minimizer of

$$\|\mathbf{Kx} - \mathbf{y}\|_2^2 + \lambda_*\|\mathbf{x}\|_2^2,$$

because the term $-\lambda\delta$ is constant. We now recognize the ordinary Tikhonov problem. Note that the problem formulated with a constraint is harder because the proper Lagrange multiplier λ needs to found via a root finding procedure. The formulation using a constraint may, on the other hand, be more natural if a bound on the solution is known.

3.1.2 Projection Methods

This section explains the different variations on projection methods. In § 3.2.2 we describe the more implementation details.

The other popular regularization method is the TSVD method. Here we directly modify the ill-posed operator into a rank deficient approximation. Using the SVD of the ill-posed operator $\mathbf{K} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, we can form a truncated operator

$$\mathbf{K}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T, \tag{3.3}$$

where $\mathbf{U}_k = [\mathbf{u}_1 \dots \mathbf{u}_k]$, $\mathbf{V}_k = [\mathbf{v}_1 \dots \mathbf{v}_k]$ and $\mathbf{\Sigma}_k = \mathrm{diag}([\sigma_1 \dots \sigma_k])$ are created from the first k columns and k first singular values respectively. The TSVD solution is the least squares solution of minimum norm to the problem with the truncated operator, that is,

$$\mathbf{x}_k = \underset{\mathbf{x}}{\mathrm{argmin}} \|\mathbf{x}\| \quad \text{subject to} \quad \min \|\mathbf{K}_k \mathbf{x} - \mathbf{y}\|_2$$

or written differently

$$\begin{aligned} \mathbf{x}_k &= \mathbf{V}_k \underset{\mathbf{z}}{\mathrm{argmin}} \|\mathbf{K}\mathbf{V}_k \mathbf{z} - \mathbf{y}\|_2 \\ &= \mathbf{V}_k \underset{\mathbf{z}}{\mathrm{argmin}} \|\mathbf{\Sigma}_k \mathbf{z} - \mathbf{U}_k^T \mathbf{y}\|_2, \end{aligned}$$

where we need to solve the projected $k \times k$ problem $\mathbf{\Sigma}_k \mathbf{z} - \mathbf{U}_k^T \mathbf{y}$. The subspace $\mathbf{V}_k$ is in some sense the best k dimensional subspace to project the solution into as information is mostly unharmed in the first components of the SVD. However, the SVD computation uses $\mathcal{O}(\min(m,n)mn)$ operations for a matrix $\mathbf{K} \in \mathbb{R}^{m \times n}$ and quickly becomes infeasible when the problem size grows. Consequently other subspaces need to be considered.

Krylov Subspace Methods

Iterative methods such as LSQR and GMRES can also be seen in a subspace setting, where a Krylov subspace (see Def. 2.10) takes the place of the SVD subspace. For example, LSQR with the zero starting vector minimizes the least squares residual $\|\mathbf{K}\mathbf{x}_k - \mathbf{y}\|_2$ with the constraint

$$\mathbf{x}_k \in \mathcal{K}_k \left(\mathbf{K}^T \mathbf{K} \,, \mathbf{K}^T \mathbf{y} \right)$$

at each step k. The GMRES method [116] for square non-symmetric problems has also been considered for regularization see [19, 82, 109] and MINRES [105] for symmetric indefinite systems has also been considered [88]. Hanke has proposed the MINRES variant mrII [57, 60] for regularization of symmetric indefinite problems. See Table 3.2 for a overview of the just mentioned Krylov methods and their choice of subspace.

The regularization of the subspace methods depends on the subspace dimension (and type). If the subspace spans the entire solution space we get the unwanted

Method	Subspace	Note
LSQR/CGLS	$\mathcal{K}_k(\mathbf{K}^T\mathbf{K}, \mathbf{K}^T\mathbf{y})$	General $\mathbf{K}$
GMRES	$\mathcal{K}_k(\mathbf{K}, \mathbf{y})$	Square $\mathbf{K}$
MINRES	$\mathcal{K}_k(\mathbf{K}, \mathbf{y})$	Square and symmetric $\mathbf{K}$
mrII	$\mathcal{K}_k(\mathbf{K}, \mathbf{K}\mathbf{y})$	Square and symmetric $\mathbf{K}$

Table 3.2: Iterative methods minimizing the residual $\|\mathbf{K}\mathbf{x}-\mathbf{y}\|_2$ in a Krylov subspace.

least squares solution. During the first iterations we (hopefully) include components unharmed by the noise but at some point we start to include components dominated by noise. In the case of TSVD this is obvious when we look at the Picard plot (Fig. 2.2). In the case of the Krylov methods the same behavior has been observed, i.e., that components belonging to large singular values are recovered first and then later components contaminated with noise appears. This has lead to the notion of "semi-convergence" where the iterative method first converges toward a nice regularized solution but then — at some point — starts to include noisy components as it converges toward the least squares solution. A more formal treatment of Krylov methods, regularization and ill-posed problems can be found in [57].

In connection with iterative methods preconditioning is often advocated to speed up convergence. Applying a preconditioner to one of the mentioned Krylov methods changes the Krylov subspace and thus the result at each step. The general preconditioners are constructed to accelerate convergence toward the exact solution, but in the case of ill-posed problems we are not interested in the exact least squares solution. Thus the preconditioner must be constructed to speed up the convergence of "good" components in the solution and not of components dominated by noise. The preconditioners described in [60, 61] and [86] take this approach.

Modified Subspace Methods

The selected subspace $\mathcal{S}$ does not always allow for certain features to be reconstructed. Hence methods that add components from the orthogonal space have been proposed. The modified TSVD (MTSVD) method [73] solves the problem

$$\min\|\mathbf{L}\mathbf{x}\|_2 \text{ subject to } \|\mathbf{K}_k\mathbf{x}-\mathbf{y}\|_2 = \min,$$

where $\mathbf{K}_k$ is the TSVD operator defined in (3.3). The TSVD operator has a non-trivial null-space in which a component of $\mathbf{x}$ is found through a minimization of $\min\|\mathbf{L}\mathbf{x}\|_2$. The piece-wise polynomial TSVD (PP-TSVD) method [70] solves

$$\min\|\mathbf{L}\mathbf{x}\|_1 \text{ subject to } \|\mathbf{K}_k\mathbf{x}-\mathbf{y}\|_2 = \min,$$

with a subtle change to a l_1-norm in the modification.

The solutions to these "modified" subspace methods can be written as a sum of two terms

$$\mathbf{x}_{\text{reg}} = \mathbf{x}_0 + \mathbf{x}_\perp,$$

where

$$\mathbf{x}_0 = \underset{\mathbf{x}\in\mathcal{S}}{\operatorname{argmin}} F_0(\mathbf{K}\mathbf{x}-\mathbf{y}) \text{ and } \mathbf{x}_\perp = \underset{\mathbf{x}\perp\mathcal{S}}{\operatorname{argmin}} F_1(\mathbf{L}(\mathbf{x}_0+\mathbf{x}))$$

are contributions from the projected problem and the orthogonal subspace respectively. The TSVD method use the subspace spanned by the first k right singular vectors as $\mathcal{S}$, $F_0(\mathbf{r}) = \|\mathbf{r}\|_2$ and $x_\perp = 0$, while the MTSVD method has $F_1(\mathbf{r}) = \|\mathbf{r}\|_2$ and the PP-TSVD method has $F_1(\mathbf{r}) = \|\mathbf{r}\|_1$. Naturally other methods can be found by combining other choices of F_0 and F_1 from Table 3.1. Villiers et al. [29] suggested a modification that not only minimized in a subspace but also enforced non-negativity of the solution, that is, to solve

$$\mathbf{x}_\perp = \underset{\mathbf{x}\perp\mathcal{S}}{\operatorname{argmin}} \|\mathbf{x}_0+\mathbf{x}\| \text{ subject to } \mathbf{x}_0+\mathbf{x} \geq \mathbf{0}$$

However, this particular modification does not always have a feasible solution at all and has not been considered further.

The subspaces can be chosen to be from the SVD or it can be created iteratively via some Krylov method, see Table 3.2. At times it can be useful to "enrich" a Krylov subspace method with specially chosen vectors approximating the solution, see [20]. The Krylov methods are useful for large-scale problems when the SVD is infeasible to compute. On the other hand the SVD based methods are theoretically better understood.

3.1.3 Hybrid Methods

Projection methods like MINRES, GMRES and LSQR can have a regularizing effect from limiting the number of iterations and thus the solution subspace. Unlike SVD-based methods they can be applied to large-scale problems—at times as the the only option. However, unwanted components may sneak in or the optimal number of iterations is unknown and to many iterations are performed. The so-called hybrid methods [59] attempt to fix these problems by adding regularization on the relatively small inner least squares problems solved implicitly by the iterative methods. The idea originates from papers of O'Leary and Simmons [104] and Björck [12].

The hybrid methods combine a subspace method with another regularization method on the projected problem. The LSQR method solves projected problems of the type

$$\mathbf{x}_k = \underset{\mathbf{x}\in\mathcal{S}}{\operatorname{argmin}} \|\mathbf{K}\mathbf{x}-\mathbf{y}\|_2, \tag{3.4}$$

where the subspace $\mathcal{S} = \mathcal{K}_k(\mathbf{K}^T\mathbf{K}, \mathbf{K}^T\mathbf{b})$. LSQR is based on Lanczos bidiagonalization to lower bidiagonal form, see Alg. 3. Let $\mathbf{V}_k = [\mathbf{v}_1 \ldots \mathbf{v}_k]$ be the orthonormal basis for the subspace $\mathcal{S}$ computed during the bidiagonalization. Now using $\mathbf{V}_k$ and the relation for the bidiagonal system and the Krylov spaces (2.15) we can reformulate the least squares problem (3.4) to

$$\begin{aligned} \mathbf{x}_k = \mathbf{V}_k\mathbf{z}_k \quad \text{where } \mathbf{z}_k &= \underset{\mathbf{z}}{\operatorname{argmin}} \|\mathbf{K}\mathbf{V}_k\mathbf{z} - \mathbf{y}\|_2 \\ \mathbf{z}_k &= \underset{\mathbf{z}}{\operatorname{argmin}} \|\mathbf{U}_k\mathbf{B}_k\mathbf{z} - \mathbf{y}\|_2 \\ \mathbf{z}_k &= \underset{\mathbf{z}}{\operatorname{argmin}} \|\mathbf{B}_k\mathbf{z} - \beta\mathbf{e}_1\|_2, \end{aligned}$$

where $\mathbf{B}_k \in \mathbb{R}^{(k+1)\times k}$. In LSQR only the most recent elements of the bidiagonal and only two of the columns of $\mathbf{V}_k$ are needed to continue the iteration giving a very space efficient algorithm. If we instead create $\mathbf{B}_k$ explicitly and store the Krylov subspace vectors we can apply some regularization method to the bidiagonal least squares problem to enforce a second level of regularization. The advantage is that k is relatively small and it is therefore possible to use the theoretically more sound SVD based methods to solve the projected system. Maybe more importantly we can use SVD based methods to find appropriate regularization parameters, see e.g. [87]. This combination of an iterative method and regularization on the inner problem is denoted *hybrid* methods.

3.2 Modules

How do we exploit the generalizations and observations made in the previous section? We will need to construct a set of common solution methods (modules) that can be combined as freely as possible.

To achieve modularity we also enforce which arguments a certain type of module gets. A call to a general regularization method only has three arguments, the operator, the right-hand side and a structure of options, see the tutorial in App. A for details and examples.

3.2.1 Penalty Methods

We will derive a general formulation for gradients and Hessians for the general penalty regularization problem in (3.2). With the gradient and Hessian available we are able to use Newton type methods for our optimization problem.

A simple example will suffice to demonstrate the structure. In our example we use only one penalty term,

$$\min F(\mathbf{x}) \text{ where } F(\mathbf{x}) = \|\mathbf{K}\mathbf{x} - \mathbf{y}\|_2^2 + \lambda^2 F_F(\mathbf{L}\mathbf{x}) = F_2(\mathbf{K}\mathbf{x} - \mathbf{y}) + \lambda^2 F_F(\mathbf{L}\mathbf{x}),$$

and $F_F(\mathbf{r})$ is the Fair penalty function, see Table 2.1.

To recapitulate our notation from §2.6 we define the residual as

$$\mathbf{r} = \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{K}\mathbf{x} - \mathbf{y} \\ \mathbf{L}\mathbf{x} \end{bmatrix}, \tag{3.5}$$

where we have extracted the inner linear arguments of F_2 and F_F. The gradient can be expressed in terms of the residual

$$\begin{aligned}\nabla_{\mathbf{x}} \left(F_2(\mathbf{Kx}-\mathbf{y}) + \lambda^2 F_F(\mathbf{Lx})\right) &= \mathbf{K}^T \nabla F_2(\mathbf{r}_1) + \lambda^2 \mathbf{L}^T \nabla F_F(\mathbf{r}_2) \\ &= \begin{bmatrix}\mathbf{K}\\ \mathbf{L}\end{bmatrix}^T \begin{bmatrix}\mathbf{I} & \\ & \lambda\mathbf{I}\end{bmatrix} \begin{bmatrix}\mathbf{I} & \\ & \lambda\mathbf{I}\end{bmatrix} \begin{bmatrix}\nabla F_2(\mathbf{r}_1)\\ \nabla F_F(\mathbf{r}_2)\end{bmatrix},\end{aligned}$$

where $\nabla F_2(\mathbf{r}) = 2[r_1\, r_2\, \cdots r_n]^T$ and $\nabla F_F(\mathbf{r}) = [f_F'(r_1)\, f_F'(r_2)\, \cdots\, f_F'(r_n)]^T$; see also §2.6. The Hessian $\mathbf{H}(F(\mathbf{x}))$ can also be expressed through the chain rule in terms of $\mathbf{r}$ and the Hessian of $F_2(\mathbf{r}_1)$ and $F_F(\mathbf{r}_2)$;

$$\begin{aligned}\mathbf{H}(F(\mathbf{x})) &= \mathbf{K}^T \mathbf{H}(F(\mathbf{r}_1))\mathbf{K} + \lambda^2 \mathbf{L}^T \mathbf{H}(F_F(\mathbf{r}_2))\mathbf{L} \\ &= \begin{bmatrix}\mathbf{K}\\ \mathbf{L}\end{bmatrix}^T \begin{bmatrix}\mathbf{I} & \\ & \lambda\mathbf{I}\end{bmatrix} \begin{bmatrix}\mathbf{H}(F_2(\mathbf{r}_1)) & \mathbf{0}\\ \mathbf{0} & \mathbf{H}(F_F(\mathbf{r}_2))\end{bmatrix} \begin{bmatrix}\mathbf{I} & \\ & \lambda\mathbf{I}\end{bmatrix} \begin{bmatrix}\mathbf{K}\\ \mathbf{L}\end{bmatrix},\end{aligned}$$

where $\mathbf{H}(F_2(\mathbf{r})) = 2\mathrm{diag}([1\,1\,\cdots\,1])$ and

$$\mathbf{H}(F_F(\mathbf{r})) = \mathrm{diag}([f_F''(r_1)\, f_F''(r_2)\, \ldots\, f_F''(r_n)])$$

are the element-wise second derivatives of functions $F_2(\mathbf{r}_1)$ and $F_F(\mathbf{r}_2)$.

With the Hessian and gradient it is straightforward to use a Newton method to find the minimizer. The Newton method solves

$$\mathbf{H}(F(\mathbf{x}_k))\mathbf{d} = -\nabla F(\mathbf{x}_k) \tag{3.6}$$

and then updates the current iterate with $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}$. The iteration can be globalized (insuring convergence for initial guesses far from the solution) by means of a line-search along $\mathbf{d}$ (used in the M$\mathcal{OO}$Re Tools toolbox) or using trust-region ideas.

The (strange) splitting of λ^2 into two multiplications with λ is done to emphasize the following relationship with a least squares problem. If we look closer at the Hessian and gradient we see that they form the normal equation system for the least squares system

$$\begin{aligned}\min_{\mathbf{d}} \Bigg\| &\begin{bmatrix}\mathbf{H}(\mathbf{F}_2(\mathbf{r}_1)) & \mathbf{0}\\ \mathbf{0} & \mathbf{H}(F_F(\mathbf{r}_2))\end{bmatrix}^{1/2} \begin{bmatrix}\mathbf{I} & \\ & \lambda\mathbf{I}\end{bmatrix} \begin{bmatrix}\mathbf{K}\\ \mathbf{L}\end{bmatrix} \mathbf{d} + \\ &\qquad \begin{bmatrix}\mathbf{H}(F_2(\mathbf{r}_1)) & \mathbf{0}\\ \mathbf{0} & \mathbf{H}(F_F(\mathbf{r}_2))\end{bmatrix}^{-1/2} \begin{bmatrix}\mathbf{I} & \\ & \lambda\mathbf{I}\end{bmatrix} \begin{bmatrix}\nabla F_2(\mathbf{r}_1)\\ \nabla F_H(\mathbf{r}_2)\end{bmatrix} \Bigg\|_2^2\end{aligned}$$

or more compactly as

$$\min_{\mathbf{d}} \left\| \mathbf{D} \begin{bmatrix}\mathbf{K}\\ \lambda\mathbf{L}\end{bmatrix} \mathbf{d} + \mathbf{z} \right\|_2^2, \tag{3.7}$$

where

$$\mathbf{D} = \begin{bmatrix} \mathbf{H}(F_2(\mathbf{r}_1)) & \\ & \mathbf{H}(F_F(\mathbf{r}_2)) \end{bmatrix}^{1/2} \quad \text{and } \mathbf{z} = \mathbf{D}^{-1} \begin{bmatrix} \nabla F_2(\mathbf{r}_1) \\ \lambda \nabla F_F(\mathbf{r}_2) \end{bmatrix}$$

Solving the least squares system either directly via a QR factorization or iteratively via LSQR improves the precision compared to Cholesky (direct) or CG (iteratively) directly on the (normal equation) Newton problem (3.6). The square roots of the Hessian matrices are easy as they are diagonal matrices. In case an element of the diagonal matrix $\mathbf{D}$ is zero, for example, when an residual element is larger than β in the Huber penalty, the corresponding equation is removed from the least squares problem.

It is straightforward to extend the above technique to an arbitrary number of penalty terms by simply extending the block structures. Thus, to solve our general Tikhonov/penalty problem we need to solve a number of least squares problems.

The M$\mathcal{OO}$Re Tools package provides functions for the penalty functions in Table 2.1. Each function returns not only its value, $F(\mathbf{r})$, but also a vector of first derivatives, $\nabla F(\mathbf{r})$, and a vector of the second derivatives, $\mathrm{diag}(\mathbf{H}(F(\mathbf{r})))$. The optimization routine can then easily set up the least squares problem, solve it, update the current solution and continue if a better solution is sought.

The LSQR algorithm is by default used to solve the least squares systems. The diagonal matrix $\mathbf{D}$ typically becomes very ill-conditioned as some elements get large and some almost zero which can lead to inaccurate solutions. The MINRES-L algorithm [15] takes special caution to solve the weighted least squares problem with a small forward error and could be considered for high precision calculations, for example, in the final stages of a Newton iteration.

Adding Non-Negativity

The non-negativity constraints arise naturally in many problems. If the result is a series of temperatures measured in Kelvin it makes sense to require a non-negative solution. Also in image reconstruction it does not make sense to have a negative number of photons. The following briefly describes the GPRN algorithm [124, Alg. 9.3.3] modified to our particular problem setup and to use a least squares solver to find the "reduced Newton step".

The example problem is

$$\begin{aligned} \min \quad & F(\mathbf{x}) \text{ subject to } \mathbf{x} \geq \mathbf{0} \\ \text{where} \quad & F(\mathbf{x}) = F_0(\mathbf{K}\mathbf{x} - \mathbf{y}) + \lambda^2 F_1(\mathbf{L}\mathbf{x}). \end{aligned}$$

Let $\mathbf{x}_k$ be the solution at iteration k. Then the active set is defined as $\mathcal{A}(\mathbf{x}_k) = \{i \mid x_i = 0\}$, that is, the set of indices of zero elements in the vector $\mathbf{x}_k$.

The *reduced gradient* is the usual gradient at the non-active indices, i.e.,

$$[\nabla_R F(\mathbf{x})]_i = \begin{cases} 0 & \text{if } i \in \mathcal{A}(\mathbf{x}) \\ [\nabla F(\mathbf{x})]_i & \text{otherwise} \end{cases}$$

and similarly the *reduced Hessian* is defined

$$H_R = [\nabla_R^2 f(\mathbf{x})]_{ij} = \begin{cases} \delta_{ij} & \text{if } i \in \mathcal{A} \text{ or } j \in \mathcal{A} \\ [\nabla^2 f(\mathbf{x})]_{ij} & \text{otherwise} \end{cases}$$

The reduced Newton search direction is

$$\mathbf{d} = -\mathbf{H}_R^{-1} \nabla_R F(\mathbf{x}) \tag{3.8}$$

Solving equation (3.8) can be seen as finding a Newton step restricted to a set of variables, in this case the non-zero components of $\mathbf{x}$. If the active set has been found the Newton method will converge quadratically to the solution as we have essentially removed the variables in the active set and effectively we work with an unconstrained problem.

Combined with the restricted Newton direction we use the projection operator

$$[\mathbf{P}(\mathbf{x})]_i = \max(x_i, 0)$$

in the line search

$$\tau = \underset{\tau > 0}{\operatorname{argmin}}\, f(\mathbf{P}(\mathbf{x} + \tau\mathbf{d}))$$

The algorithm alternates between restricted Newton direction steps and simple steepest decent steps, that is, steps in the direction $\mathbf{d} = -\nabla F(\mathbf{x})$. For further details see [124].

Problems with Non-Differentiable Functions

The usual l_2-norm Tikhonov problem leads to a single least squares problem and the Newton method converges in just one step. However, if we want an l_1-norm solution we run into a non-differentiable function for residual elements $r_i = 0$ and the Hessian is all zero rendering the Newton method unusable. Also in other cases the second derivative can pose a problem, for example, if $p \approx 1$, Li [91] avoided the problem by modifying the line search strategy to stop just short of a minimum with a zero residual.

In the special case of an all 1-norm problem

$$\min \left\| \begin{bmatrix} \mathbf{K} \\ \lambda\mathbf{L} \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} \right\|_1^1 = \min \|\widehat{\mathbf{K}}\mathbf{x} - \widehat{\mathbf{y}}\|_1^1$$

we can rewrite it into a linear programming (LP) problem in standard form by splitting residual vector $\mathbf{r} = \mathbf{r}_+ - \mathbf{r}_-$ and solution vector $\mathbf{x} = \mathbf{x}_+ - \mathbf{x}_-$ into vectors of purely non-negative values giving

$$
\begin{aligned}
\min\,[\mathbf{1}\,\mathbf{1}\,\mathbf{0}\,\mathbf{0}]\cdot[\mathbf{r}_+\,\mathbf{r}_-\,\mathbf{x}_+\,\mathbf{x}_-]\ \text{s.t.} \\
\begin{bmatrix}\mathbf{I} & -\mathbf{I} & \widehat{\mathbf{K}} & -\widehat{\mathbf{K}}\end{bmatrix}\begin{bmatrix}\mathbf{r}_+\\ \mathbf{r}_-\\ \mathbf{x}_+\\ \mathbf{x}_-\end{bmatrix} = \widehat{\mathbf{y}} \\
\mathbf{r}_+,\mathbf{r}_-,\mathbf{x}_+,\mathbf{x}_- \geq \mathbf{0},
\end{aligned}
$$

and apply one of many LP solvers. In this case a non-negativity constraint actually simplifies the problem by reducing the number of variables and the size of the constraint matrix. Other linear constraints are also straightforward to include in this formulation.

Solvers for linear programming problems have improved tremendously since the simplex algorithm appeared due to the many commercial applications of LP. The solvers can solve problems with thousands of unknowns and constraints, but usually they assume that the constraint matrix is sparse except for a couple of columns. If we are solving an inverse problem coming from an integral equation (as all problems but the INTERPOLATE test problem do) we get a dense operator $\mathbf{K}$. Furthermore $\mathbf{K}$ might not be directly available in matrix representation. This renders most of the available LP solvers unusable in our context. M$\mathcal{OO}$Re Tools provides an interface to Matlab's `linprog` solver for small-scale problems. A preliminary implementation of an interior method based on Mehrotra's Predictor-Corrector algorithm (see [95, 128]) is included for larger problems. Another option is to approximate the all-l_1-norm problem via the the Huber penalty function letting $\beta \to 0$ (see [93]).

3.2.2 Projection Methods

This section is formulated in terms of the equation system $\mathbf{r} = \mathbf{K}\mathbf{x} - \mathbf{y}$ as the linear part of the modification terms can be rewritten

$$\mathbf{L}(\mathbf{x}+\mathbf{x}_\perp) = \underbrace{\mathbf{L}}_{\mathbf{K}}\underbrace{\mathbf{x}_\perp}_{\mathbf{x}} + \underbrace{\mathbf{L}\mathbf{x}}_{-\mathbf{y}}$$

The basic objective in the projection methods is to minimize in a specific subspace

$$\mathbf{x} = \operatorname*{argmin}_{\mathbf{x}\in\mathcal{S}} F(\mathbf{K}\mathbf{x}-\mathbf{y}), \tag{3.9}$$

and (usually in the optional modification stage) to minimize with orthogonality constraints

$$\mathbf{x}_\perp = \operatorname*{argmin}_{\mathbf{x}\perp\mathcal{S}} F(\mathbf{K}\mathbf{x}-\mathbf{y}). \tag{3.10}$$

The two problems in (3.9) and (3.10) are basically of the same type; minimize a convex function in a certain subspace. Depending on the specific subspace it can be more convenient to express the problem as lying *in* a subspace and at other times it is better to express the solution as being *orthogonal* to a subspace. Hence it seems reasonable to have modules for both tasks.

We will not consider minimization in a Krylov space with the l_2 norm as Krylov subspace methods are designed to compute these solutions.

If we have a (preferable orthogonal) matrix $\mathbf{S} \in \mathbb{R}^{m \times n}$ with (preferably orthonormal) columns spanning the subspace $\mathcal{S}$, then the problem in (3.9) reduces to an simple unconstrained problem

$$\mathbf{x} = \mathbf{S}\mathbf{z}, \quad \text{where} \quad \mathbf{z} = \operatorname*{argmin}_{\mathbf{z}} F(\mathbf{K}\mathbf{S}\mathbf{z} - \mathbf{y}),$$

which can be solved by the method described in §3.2.1, that is, the solution is found with a series of Newton steps, where the search direction in each step is found from a least squares problem.

The minimization problem in the orthogonal subspace is on the other hand seems less simple. We have two options

$$\begin{aligned}\mathbf{x}_\perp &= \operatorname*{argmin}_{\mathbf{x}} F(\mathbf{K}(\mathbf{I} - \mathbf{S}\mathbf{S}^\dagger)\mathbf{x} - \mathbf{y}) \\ &= \operatorname*{argmin}_{\mathbf{x}_\perp} F(\mathbf{K}\mathbf{x}_\perp - \mathbf{y}) \text{ subject to } \mathbf{S}^T\mathbf{x}_\perp = \mathbf{0},\end{aligned}$$

where the first option is nice if $\mathbf{S}\mathbf{S}^\dagger = \mathbf{S}\mathbf{S}^T$, i.e., if $\mathbf{S}$ has orthonormal columns. Note, that the resulting coefficient matrix $\mathbf{D}\mathbf{K}(\mathbf{I} - \mathbf{S}\mathbf{S}^\dagger)$ of the weighted least squares system is rank deficient and the solver must be able to handle this situation.

Linear Equality Constraints

The second option might prove more simple if the subspace is not described by an orthogonal operator. The unconstrained problem can be solved with a Newton method where search directions are found via a series of weighted least squares problems as seen in §3.2.1. If we furthermore constrain each of the weighted least squares problems in (3.7) so that the search direction in each step fulfills the equality constraints, we get in notation similar to (3.7)

$$\mathbf{d} = \operatorname*{argmin}_{\mathbf{x}} \|\mathbf{D}\mathbf{K}\mathbf{x} - \mathbf{z}\|_2^2 \text{ subject to } \mathbf{S}^T\mathbf{x} = \mathbf{0},$$

we get search directions in the feasible space. In our particular setting we have a feasible starting guess $\mathbf{x} = \mathbf{0}$ for the iteration.

In order to solve an equality constrained least squares problem we form the Lagrangian

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = \|\mathbf{D}(\mathbf{K}\mathbf{x} - \mathbf{z})\|_2^2 - \boldsymbol{\lambda}^T(\mathbf{S}^T\mathbf{x}),$$

where $\boldsymbol{\lambda}$ is a vector of Lagrange multipliers. From the Lagrangian function we get the KKT first order conditions

$$\nabla\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = \begin{bmatrix} (\mathbf{D}\mathbf{K})^T(\mathbf{D}\mathbf{K}\mathbf{x} - \mathbf{z}) + -\mathbf{S}\boldsymbol{\lambda} \\ -\mathbf{S}^T\mathbf{x}_\perp \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}.$$

Introducing the residual $\mathbf{r} = \mathbf{DKx} - \mathbf{z}$ yields the symmetric *indefinite* system

$$\begin{bmatrix} \mathbf{0} & \mathbf{K}^T\mathbf{D} & -\mathbf{S} \\ \mathbf{DK} & -\mathbf{I} & \mathbf{0} \\ -\mathbf{S}^T & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{r} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{z} \\ \mathbf{0} \end{bmatrix}. \tag{3.11}$$

Direct methods for systems of this kind are surveyed in [46, Sec 4.4] while MINRES and SYMMLQ [105] are suitable iterative solvers.

Having modules to minimize in a given norm in a subspace or its orthogonal subspace enables us to solve the problems in the subspace category.

3.2.3 Hybrid Methods

We have created three hybrid modules based on Lanczos tridiagonalization (Alg. 1), the Arnoldi process (Alg. 2) and Lanczos bidiagonalization to lower form (Alg. 3) respectively.

The Lanczos bidiagonalization algorithm which also lies behind the LSQR method iteratively computes for a matrix $\mathbf{K} \in \mathbb{R}^{m\times n}$ orthogonal matrices $\mathbf{V}_k \in \mathbb{R}^{n\times k}$,$\mathbf{U}_{k+1} \in \mathbb{R}^{m\times k+1}$ and a lower bidiagonal matrix $\mathbf{B} \in \mathbb{R}^{k+1\times k}$ so that

$$\mathbf{KV}_k = \mathbf{U}_{k+1}\mathbf{B}_k.$$

If we insert it into the least squares problem $\min\|\mathbf{KV}_k\mathbf{z} - \mathbf{y}\|_2$ we get

$$\min \|\mathbf{U}_{k+1}\mathbf{B}_k\mathbf{z} - \mathbf{y}\|_2 = \min \|\mathbf{B}_k\mathbf{z} - \mathbf{U}_{k+1}^T\mathbf{y}\|_2.$$

In LSQR $\mathbf{U}_{k+1}\mathbf{y} = \mathbf{e}_1\|y\|_2$ because $\mathbf{y}$ is selected as the starting vector of the bidiagonalization. However, we will allow for other starting vectors and the Lanczos tridiagonalization hybrid method can thereby compute both a MINRES like (starting vector $\mathbf{y}$) and an mrII like (starting vector $\mathbf{Ky}$) hybrid method. The hybrid method puts the least squares system through some regularization routine like TSVD or Tikhonov, but any regularization method can be applied, even another hybrid method.

The literature has mainly used TSVD [12, 104], but other ideas have been investigated by O'Leary and Kilmer [87]. The M$\mathcal{OO}$Re Tools toolbox allows for any regularization method to do the inner regularization, including an hybrid method.

3.3 Parameter-Choice Methods

Just as important as solving the regularization problem is the choice of regularization parameter(s). If the regularization parameters are chosen inappropriately even the best and fastest algorithm known to man returns a useless result.

The subspace methods have one regularization parameter, namely the subspace dimension k corresponding to the iteration count for iterative methods. The penalty methods on the other hand can have multiple regularization parameters—one for each penalty term. The hybrid methods need at least two parameters; the subspace dimension and one or more parameter for the inner regularization problem.

The multiple regularization parameter case has not been considered here, but the interested reader may consult [7] and [17] for recent developments within this area.

We have three major parameter-choice methods which in the following are described with respect to the Tikhonov problem.

- The (Morozov) discrepancy principle [98] selects the regularization parameter so that the model fit $\|\mathbf{Kx} - \mathbf{y}\|_2$ is equal to an upper bound on the error δ, that is,

  $$\|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2 = \delta_e \text{ where } \|\mathbf{e}\|_2 \leq \delta_2.$$

  If we know that the norm of the noise $\|\mathbf{e}\|_2 = \delta$ (which we call the noise level) it does not make sense to ask for a solution $\mathbf{x}_\lambda$ where $\|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2 < \delta$ as we would fit information not present in the data $\mathbf{y}$.

  The iterative methods MINRES, GMRES and LSQR all have monotonically decreasing residuals and iterations can be stopped when the residual norm passes δ_e, making the discrepancy principle a perfect choice for these methods in case we know the noise-level.

  The residual norms of Tikhonov regularized solutions are also monotonic with respect to the regularization parameter λ. Thus the discrepancy principle is well defined for Tikhonov regularization and can be found with a root finding procedure.

- Generalized cross validation (GCV) [125] is based on statistical considerations and does not use information about the noise-level.

  Instead it minimizes the GCV function

  $$G(\lambda) = \frac{\|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2^2}{(\mathrm{trace}(\mathbf{I} - \mathbf{K}\mathbf{K}_\lambda^{\#}))^2},$$

  where $\mathbf{x}_\lambda$ is the regularized solution and $\mathbf{K}_\lambda^{\#}$ is the inverse regularized operator, that is, $\mathbf{K}_\lambda^{\#} = (\mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{I})^{-1}\mathbf{K}^T$ for standard-form Tikhonov regularization.

  The GCV function is somewhat difficult to compute for iterative methods like CG as the regularized inverse $\mathbf{K}^{\#}$ is not unique and depends not only on the iteration number (and regularization parameter) k but also on the particular right hand side, see [66, Sec. 7.4] for an overview of combining GCV and CG methods.

- The L-curve method is based on heuristic observations and is already mentioned (but not used to determine any parameters) by Lawson and Hanson [90] in 1974, but is later directly used as a parameter-choice method and named by Hansen [65]. See also [72].

  The L-curve heuristic attempts to balance the penalty $\|\mathbf{x}_\lambda\|_2$ versus the model fit, $\|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2$ for regularized solutions $\mathbf{x}_\lambda$ with regularization parameter λ.

The L-curve name comes from the characteristic shape of the curve $((\log\|\mathbf{Kx}_\lambda - \mathbf{y}\|_2, \log\|\mathbf{x}_\lambda\|_2)$, as seen on Figs. 3.1 and 3.3. Functions other than log, such as the square root, are also used but we will limit the discussion to the logarithmic scaling. A too small λ yields a large penalty term and a small model fit norm while a too large λ gives a small penalty term and a poor fit. The idea is to find a balance of the two, e.g., at the corner of the "L".

However to find a useful algorithm we need a more precise definition of the corner and several have been proposed. Hansen and O'Leary [72] suggested using the point of maximum curvature. This approach is invariant to scaling of the equations, but the computation needs derivatives of the penalty and residual fit functions. Regińska [110] suggested minimizing $\|\mathbf{Kx}_\lambda - \mathbf{y}\|_2^\beta \|\mathbf{x}_\lambda\|_2$, where $\beta > 0$. For $\beta = 1$ it corresponds to a rotation of the L-curve 45 degrees counter clockwise and finding a minimum, see Fig. 3.2. We will use this simple formulation in later experiments.

Of the three only the discrepancy principle requires knowledge or estimation of the noise-level, but the noise-level is not always available and an estimation may be unreliable. The GCV function and L-curve method on the other hand do not require the noise-level. An experimental comparison of how well λ is determined compared to the optimal regularization parameter is done in [66].

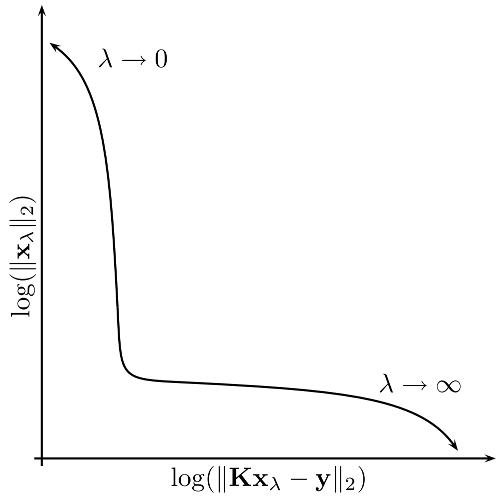


Figure 3.1: Stylized L-Curve

Large-Scale Parameter Selection

Finding a good regularization parameter for Tikhonov using the above mentioned methods can be quite difficult if the SVD is unavailable. The SVD makes it possible to evaluate the norms and derivatives for several regularization parameters fast.

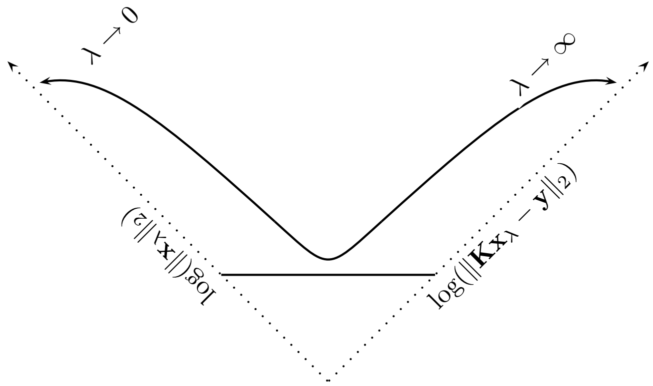
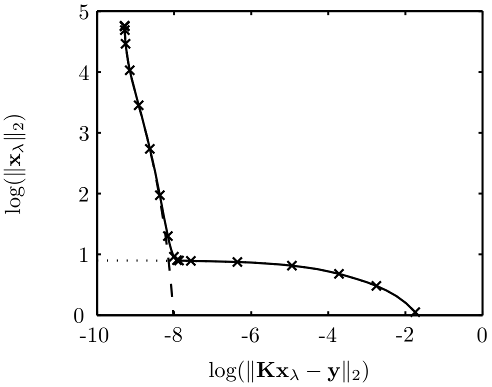
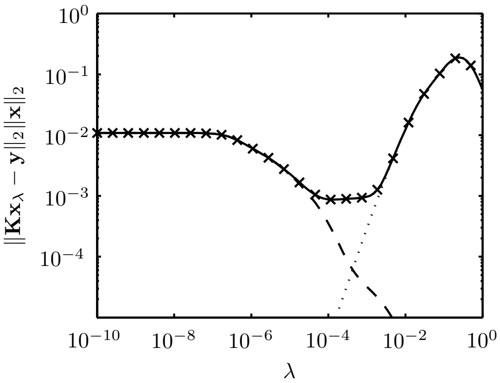


Figure 3.2: Stylized rotated L-Curve



(a) Ordinary L-Curve



(b) Rotated L-Curve

Figure 3.3: L-curve for the HEAT test problem with $\mathbf{K} \in \mathbb{R}^{100\times100}$ and white noise added. The dotted line shows the L-curve for a no noise right hand side, the dashed line is the L-curve for a pure noise right hand side and the solid line is the actual L-curve for a noisy right hand side. The x markers show points for evenly spaced regularization parameters λ. Note that the points in the L-curve do not depend linearly with λ. Thus the plot in (b) is both rotated and then "stretched" compared to (a).

Otherwise, we can approximate, e.g. the L-curve, by solving multiple systems with different parameters and thereby finding points on the L-curve. The discrepancy principle requires some kind of root finding procedure and not having the SVD significantly handicaps the process.

Especially the GCV function is very hard to compute for large problems, due to the trace term in the denominator. However, the trace term can be approximated using Hutchinson's trace estimator [80],

$$\mathrm{trace}(\mathbf{A}) \approx \mathbf{u}^T \mathbf{A} \mathbf{u},$$

where $\mathbf{u}$ is a vector with random elements -1 and 1 with equal probability $1/2$. Using an approximation gives a "Monte Carlo" GCV method; see also [42, 43] for other similar trace estimators.

A general solution technique related to the hybrid methods is to estimate the parameter from a problem projected into a smaller space where an SVD is easily obtainable; see [14, 87].

A somewhat related approach exploits a fascinating connection between the Lanczos tri/bi-diagonalization methods and Gauss-quadrature formulas. This connection enables computation of upper and lower bounds on several terms used in the parameter-choice methods for standard-form Tikhonov regularization. More details on these bounding methods can be found later in Sec. 5.1, where we consider a stopping criterion ensuring that the computed upper and lower bounds are close.

Non-l_2-norm Problems

If we do not use l_2-norm squared for residuals and penalty functions the problem gets more problematic as we are unable to utilize many of the tools provided for the l_2-norm situation. The approximations all depend on the penalty functions being of the least squares type and cannot be applied to problems with other penalty functions. However, the parameter found in a least squares setting might perform well as at least a starting guess for other norms.

3.4 Combining Modules

At this point we have considered which modules to implement. We will now consider the arguments passed to the regularization routines so that modules are easily exchanged. We can do that by enforcing a default number and order of arguments to the routines.

We have decided that all modules for standard regularization use the following form

```
[X, extra] = regalgorithm(K, y, options)
```

The operator and right-hand side is passed as the two first options while all options such as regularization parameters, tolerance criteria and which data to return are passed through the `options` structure. The regularized result is returned in `X` while extra information collected during computations can be found in the `extra` structure. The extra information could be residual and penalty norms from each iteration of an iterative algorithm, but also a collection of vectors describing the solution space for the Krylov methods.

The results shown in the following are computed with the M$\mathcal{OO}$Re Tools in Matlab and the commands used to generate them are listed in the tutorial in App. A.

3.4.1 Combining Modules — Numerical Experiments

This section demonstrates the results of using the modules presented. We demonstrate both some well-known regularization approaches as well as some exotic variations. Appendix A.3 shows how the M$\mathcal{OO}$Re Tools package has been used to create the plots shown in the following.

3.4.2 PP-LSQR

The PP-TSVD [70] method solves the modified subspace problem

$$\begin{aligned} \mathbf{x} &= \mathbf{x}_0 + \mathbf{x}_\perp \quad \text{where} \\ \mathbf{x}_0 &= \operatorname*{argmin}_{\mathbf{x}\in\mathcal{V}} \|\mathbf{K}\mathbf{x} - \mathbf{y}\|_2, \\ \mathbf{x}_\perp &= \operatorname*{argmin}_{\mathbf{x}\perp\mathcal{V}} \|\mathbf{L}(\mathbf{x} + \mathbf{x}_0)\|_1 \end{aligned}$$

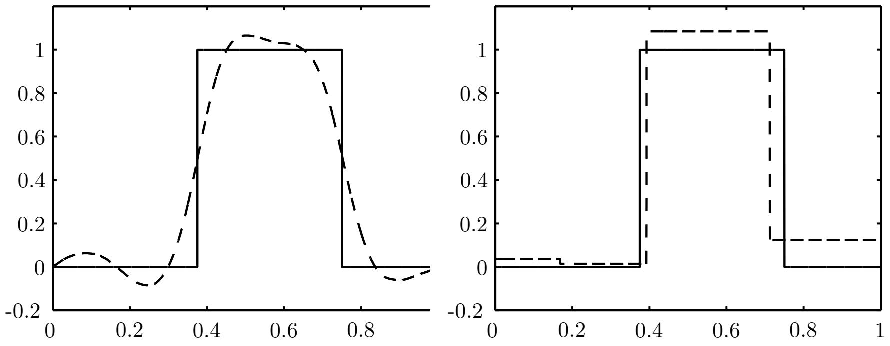
and $\mathcal{V}$ is a subspace spanned by the k first singular vectors. The PP-LSQR method exchanges the SVD basis with the Krylov space

$$\mathcal{V} = \mathcal{K}_k(\mathbf{K}^T\mathbf{K}, \mathbf{K}^T\mathbf{y}),$$

computed iteratively via the LSQR method. If $\mathbf{K}$ is (too) large and without structure so that an SVD cannot be computed the PP-LSQR method might show useful, if PP-TSVD like features are wanted. The LSQR and PP-LSQR solutions after 5 iterations are shown in Fig. 3.4. The 5 LSQR iterations used to find the subspace $\mathcal{V}$ (and the LSQR solution) for this particular 4000×4000 problem takes approximately 1 second, while the full SVD of $\mathbf{K}$ takes almost 2 hours to compute on the same computer—methods to find only the first SVD components should be faster but have not been tried. The modification step, that is, setting up and solving the LP-problem via `linprog` from the Optimization Toolbox, uses 14 seconds for this example.

3.4.3 A GMRES-Tikhonov Hybrid Method

The hybrid methods have usually been implemented via the LSQR algorithm. However, the GMRES method should also be considered for square systems. Furthermore,



(a) Solution (solid) and LSQR (dashed)

(b) Solution (solid) and PP-LSQR (dashed)

Figure 3.4: LSQR and PP-LSQR solutions for a problem with a piecewise constant solution. The test problem is DERIV2 with $\mathbf{K} \in \mathbb{R}^{4000\times4000}$, a piece-wise continuous solution and a right-hand side with normal distributed noise with norm $10^{-2}\|\mathbf{y}\|$. The LSQR solution is very smooth after 5 iterations. The "PP" modification with $\mathbf{L} = \mathbf{L}_1$, an approximation of the first derivative, recreates the discontinuous nature of the solution.

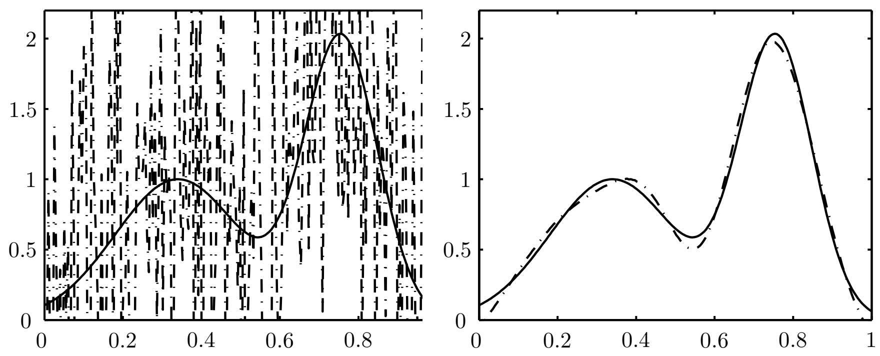
GMRES is the only option if the operator is not symmetric and multiplication with the transposed operator is not possible.

We choose Tikhonov regularization with the GCV parameter choice strategy as the inner regularization method. In Fig. 3.5 we see that too many iterations (in this example only one too many) introduce noise in the solution. Introducing regularization on the inner problem saves the day and the noisy components are damped damped. The extra work is not overwhelming as the inner regularization step only uses the SVD of a 10×9 upper Hessenberg matrix and the GCV parameter choice method is able to efficiently compute the regularization parameter using this small SVD. Similar experiments with the GMRES-Tikhonov method and image restoration can be found in the Master's Thesis [82].

3.4.4 Robust TSVD

To demonstrate how the "robust" methods perform we created a test problem with normally distributed noise except for three (extreme) outliers in the right-hand side, see Fig. 3.6.

The TSVD method is formulated via the l_2-norm, which we know is sensitive with respect to outliers in the data. Using a robust penalty function instead of the least squares penalty we can to some degree remove the influence of the outliers on the solution. Figure 3.7 compares the solutions of a standard l_2 TSVD and a "Fair



(a) Solution (solid) and GMRES after 9 iterations (dashed)

(b) Solution (solid) and Hybrid GMRES after 9 iterations (dashed)

Figure 3.5: The GMRES and hybrid GMRES-Tikhonov methods on the SHAW test problem with $\mathbf{K} \in \mathbb{R}^{200\times 200}$. Normal distributed noise with norm $10^{-3}\|\mathbf{y}\|_2$ is added to the right-hand side. The GMRES solution explodes in the 9th iteration (one to many). The hybrid method uses Tikhonov regularization with the GCV parameter choice method.

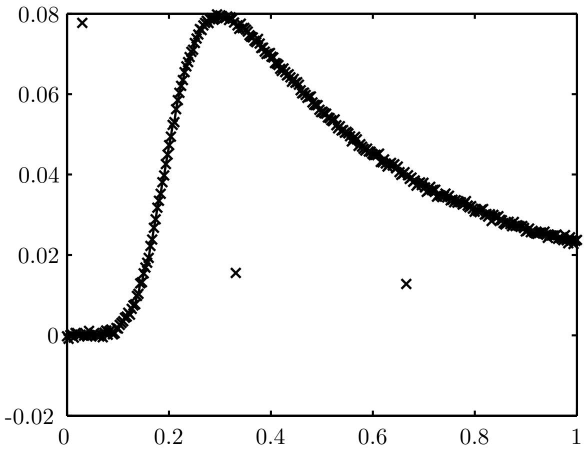
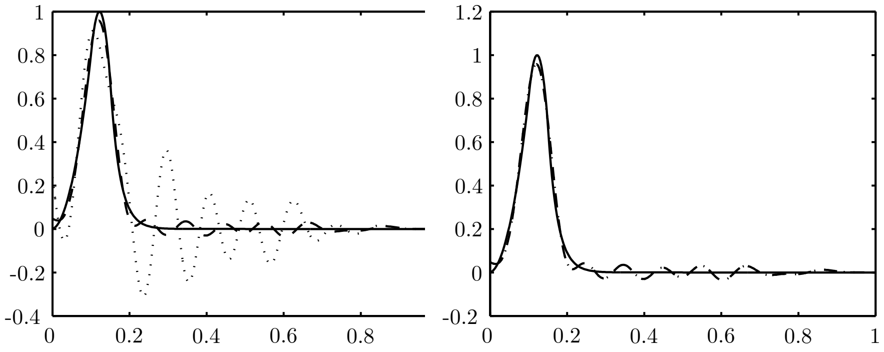


Figure 3.6: Right-and side of problem with outliers. The test problem is HEAT with $n = 300$. Normally distributed noise with norm $\|\mathbf{e}\|_2 = 10^{-2}\|\mathbf{y}\|$ is added to the solution. Finally outliers were added at index 10, 100 and 200. This particular right-hand side is used in Figs. 3.7, 3.9 and 3.10.

TSVD" method,

$$\mathbf{x}_{\mathrm{reg}} = \operatorname*{argmin}_{\mathbf{x}} F_F(\mathbf{K}_k\mathbf{x} - \mathbf{y}).$$

If the right-hand side does not have any outliers they give approximately the same solution. If we solve with the right-hand side with outliers the l_2 TSVD solution starts to oscillate while the TSVD using the Fair penalty is almost unharmed.



(a) Normal l_2 TSVD (b) TSVD with Fair penalty

Figure 3.7: Standard and robust TSVD with truncation parameter $k = 20$ applied to problems with outliers (dotted solution) and without outliers (dashed solution). The test problem is described in Fig. 3.6 and is also solved via variations of Tikhonov in Figs. 3.9 and 3.10.

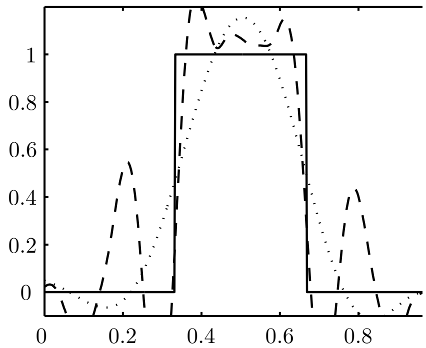
3.4.5 A PP-LSQR-TSVD Hybrid Method

To illustrate the modularity and flexibility we now try out a combination of a hybrid method using a projection method as inner regularization and a modification. The LSQR Krylov subspace method is selected and we use TSVD regularization on the inner problem. We use GCV to determine the regularization parameter for the inner Tikhonov problem. Finally, we modify the solution according to the PP scheme, that is, we add find the solution from $\mathbf{x}_{\mathrm{reg}} = \mathbf{x}_0 + \mathbf{x}_\perp$ where

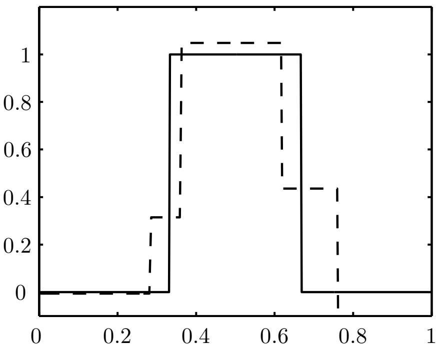
$$\mathbf{x}_\perp = \operatorname*{argmin}_{\mathbf{x} \perp \mathcal{K}_k(\mathbf{K}^T\mathbf{K}, \mathbf{K}^T\mathbf{y})} \|\mathbf{L}_1(\mathbf{x} + \mathbf{x}_0)\|_1,$$

$\mathbf{x}_0$ is the solution obtained from the Hybrid LSQR-TSVD method, $\mathbf{L}_1$ is an approximation to the first derivative, and $\mathcal{K}_k\left(\mathbf{K}^T\mathbf{K}, \mathbf{K}^T\mathbf{y}\right)$ is the k dimensional subspace that the hybrid solution lies in.

Fig. 3.8(a) shows an example of LSQR with and without inner TSVD regularization. We see that the inner regularization compensates somewhat for iterating



(a) Solution (solid), LSQR (dotted) and hybrid LSQR (dashed)



(b) Solution (solid) and hybrid LSQR with modification (dashed)

Figure 3.8: LSQR, hybrid LSQR and hybrid LSQR with modification—all after 7 iterations. The example uses the HEAT problem with $\mathbf{K} \in \mathbb{R}^{400\times 400}$, a piecewise constant solution and normal distributed noise with norm $10^{-2}\|\mathbf{y}\|_2$ is added to the right hand side.

too far as the oscillations are damped. Similar to the PP-LSQR method illustrated previously we observe from Fig. 3.8(b) that the modification step enables the reconstruction of sharp edges.

3.4.6 A Robust Tikhonov Method

If the right hand side contains measured data with a possibility of outliers it is possible to "automatically ignore" those by using a robust estimator instead of the usual l_2-norm to measure the residual. In this example we will use the Fair penalty

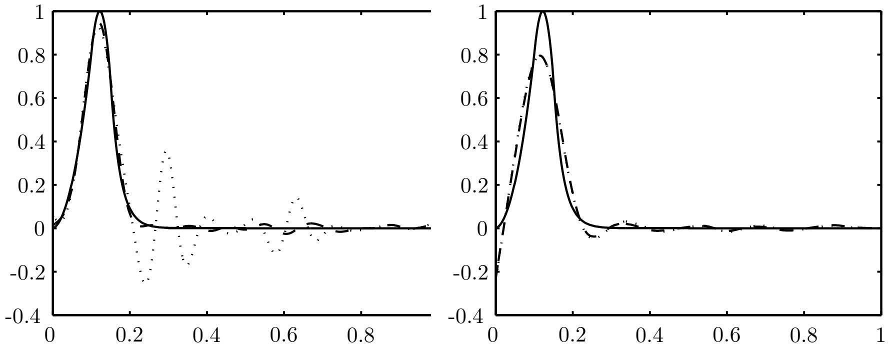
$$F_F(\mathbf{r}) = \sum_{i=1} n f_\beta(r_i), \text{ where } f_\beta(z) = \beta^2(|z|/\beta - \log(1 + |z|/\beta)).$$

We use the Fair penalty on the model fit and the l_2-norm on the penalty term, that is, we are minimizing

$$\min_{\mathbf{x}} \left\{ F_F(\mathbf{Kx} - \mathbf{y}) + \lambda^2 F_2(\mathbf{L}_2\mathbf{x}) \right\},$$

where $\mathbf{L}_2$ is an discrete approximation to the second derivative. From Fig. 3.9(a) we observe that the l_2-norm Tikhonov solution "stretches" far to catch the outliers while the Fair Tikhonov solution seen in Fig. 3.9(b) is almost unharmed by the outliers.

The HEAT test problem is dealing with temperatures. If we assume that the measuring unit is degrees Kelvin it does not make sense to have negative temperatures. If we consequently add non-negativity constraints to the solution we get the result



(a) Solution (solid), 2-norm Tikhonov solution without outliers (dashed) and with outliers (dotted)

(b) Solution (solid) and Fair penalty Tikhonov solution without outliers (dashed) and with outliers (dotted)

Figure 3.9: Robust Tikhonov regularization solutions. The test example is HEAT, $\mathbf{K} \in \mathbb{R}^{300\times 300}$. White noise $\|\mathbf{e}\|_2 = 10^{-2}\|\mathbf{y}\|$ and three outliers are added to the right-hand side, see Fig. 3.6. The regularization matrix is $\mathbf{L}_2$ and the regularization parameter is $\lambda = 10^{-1}$ in both cases and has not been optimized.

shown in Fig. 3.10. In Fig. 3.10(a) we use the 2-norm and the outliers disturb the solution around 0.3, but not to the same extent as without the outliers. Introducing the Fair penalty function on the model fit does improve the solution, cf. Fig. 3.10(b) where we do not see the extra peak around 0.3.

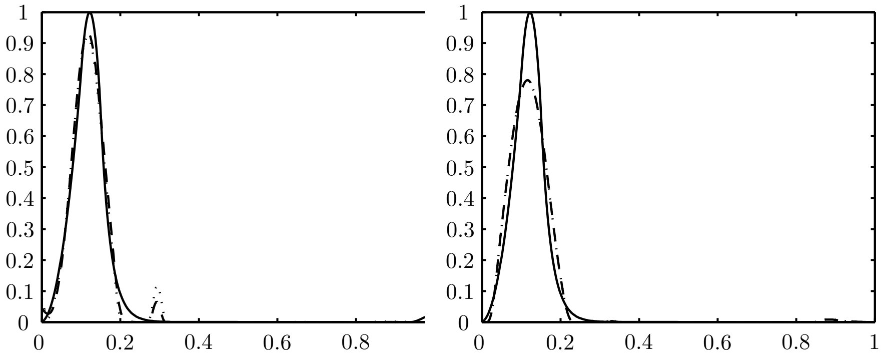
3.4.7 L-Curves for the Fair Problem

To get an idea whether we have something like an L-curve when we use other norms we create a plot of $(\log(F_F(\mathbf{K}\mathbf{x}_\lambda - \mathbf{y})), \log(\|\mathbf{x}_\lambda\|_2^2))$ where we compute solutions to

$$\mathbf{x}_\lambda = \operatorname*{argmin}_{\mathbf{x}}\{F_F(\mathbf{K}\mathbf{x} - \mathbf{y}) + \lambda\|\mathbf{x}\|_2^2\} \tag{3.12}$$

for λ in the interval $[10^{-12}; 1]$.

Figure 3.11 shows a distinct corner which is close to the optimal choice of regularization parameter. As the L-curve is based on a heuristic argument it is reasonable that we can use the L-curve also when the penalties are not l_2-norms. Indeed the L-curve is also found for non-linear ill-posed problems, as seen in [55]. Finding the corner is, on the other hand, not an easy task as many of the shortcuts available in the l_2-norm case are not available. For example, we cannot utilize the SVD to diagonalize the problem. The "minimum distance function" presented in [7] finds the corner via a fixed point iteration.



(a) Solution (solid), 2-norm Tikhonov solution without outliers (dashed) and with outliers (dotted)

(b) Solution (solid), Fair Tikhonov solution without outliers (dashed) and with outliers (dotted)

Figure 3.10: Robust Tikhonov with non-negativity constraints. The problem is the same as in Fig. 3.9.

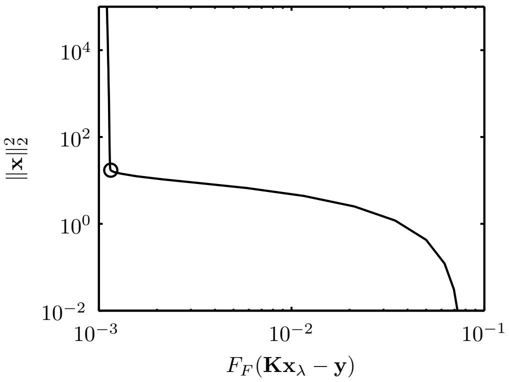


Figure 3.11: L-Curve with Fair penalty on model fit and l_2-norm penalty on solution norm (see (3.12)). The right-hand side is illustrated in Fig. 3.6.

3.5 Summary

We have seen how a number of known regularization algorithms can be split into independent modules. The hybrid methods are by construction modular. The projection methods also have a natural modular formulation as the optional modification step is separate from the actual projection step.

On the other hand we are not able to split the penalty type methods into modules as such. Instead we have created one algorithm that can solve penalty methods with one or more penalty terms of different type where the penalty functions take the role as modules.

We have seen that problems where we use norms other than the 2-norm can be solved through a series of least squares problems. Additionally we can constrain the solution with linear equality and non-negativity constraints. As we do not assume anything about the operators involved in the least squares problems we default to using LSQR to solve the unconstrained and linear equality constrained problems. The non-negativity problems can be solved solved with a projected Newton type method.

The parameter choice methods are difficult to modularize as they are very dependent on the actual regularization method used. Hence separate groups of modules are needed for each kind of regularization algorithm. We need at least one for algorithms with a discrete regularization parameter like the subspace methods and one group for the regularization methods with continuous regularization parameter like the penalty functions.

CHAPTER 4

Object Oriented Programming

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. (…) Experienced designers evidently know something inexperienced don't. What is it? Gamma, Helm, Johnson and Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software" [41].

With the advent of object-oriented languages like C++ and Java, computer science moved to a new still-higher level of programming and design. There's no turning back. An object-oriented approach (OO for short) is the way. Rudy Rucker, "Software Engineering and Computer Games" [115].

In this chapter we introduce the *object oriented* (OO) programming languages along with a brief note on the procedural and functional programming languages that are the main alternatives.

Having introduced OO programming in general we justify using Matlab as the language for the M$\mathcal{OO}$Re Tools toolbox. As shown in the previous chapter most of our problems reduce to problems of solving least squares problems of various forms and we look how to represent the matrices and vectors in the linear algebra setting using OO techniques.

Using the extra abstraction layer that OO provides eases many tasks but also implies penalties with respect to running time and memory usage.

4.1 Programming Paradigms

Making programs can be hard, but making code that is easy to maintain and extend is definitely hard, especially when the size of the program grows. Without careful planning and detailed specifications of the program it quickly becomes impossible to implement anything but the simplest program. As a consequence, different approaches to go from some problem through specification to an actual program exist and a plethora of programming languages have been created to aid the programmer to create correct programs faster. We will consider three approaches including the OO approach in the following and argue why we use the OO approach.

Imperative/Procedural Programming

The "imperative or procedural" programming languages assist the programmer in decomposing problems into a number of preferably small and manageable functions. A group of related functions can then be packaged together in libraries or toolboxes.

Often the procedural languages allow for direct manipulation of pointers and memory and there is a close correspondence in structure between the procedural language and how the machine works. These features have made these languages popular for people aiming at the best possible performance as they to a large extent are able to exploit the properties of the particular computer architecture. Also the close correspondence between language and the machine simplified the compilers and most of the early languages such as Fortran 77 [96], Pascal [127] and C [85] are all inherently procedural programming languages.

Functional Programming

The functional programming languages define everything with functions in the mathematical sense, that is, if $a = 4$ we cannot later assign another value ($a \leftarrow 5$ in our notation), see [79]. In procedural languages it is common that variables change value; just think of the loop variables.

Furthermore "side effects" are not allowed, that is, a function cannot change anything outside its scope. A pure functional programming language has no variables, no assignments and no iterations in the usual sense. Everything is done through (often recursive—to get the loops) *function* evaluation. These features make proof of correctness for the programs feasible and functional programming languages are very useful in discussing computer science problems such as determining whether a language is proper.

Examples of functional languages are LISP [118], Haskell [83] and Standard ML [97]. Within scientific computing and linear algebra these languages have never been widely used, as they have a reputation of being slow compared to more familiar languages such as Fortran and C. The familiarity aspect is most likely a major factor. Loops are common constructs in numerical algorithms, but they are somewhat "obfuscated" in function languages. For example, the matrix-matrix multiplication can be written as follows in Standard ML (extracted from [108]):

```
fun headcol [] = []
 |  headcol ((x::_) :: rows) = x :: headcol rows;

fun tailcols [] = []
 |  tailcols ((_::xs) :: rows) = xs :: tailcols rows;
```

```
fun transp ( []::rows ) = []
 | transp rows  = headcol rows :: transp (tailcols rows);

fun dotprod ( [], [] ) = 0.0
 | dotprod ( x::xs, y::ys ) = x*y + dotprod(xs, ys);

fun rowprod ( row, [] ) = []
 | rowprod ( row, col::cols ) =
              dotprod(row, col) :: rowprod(row, cols);

fun rowlistprod ( [], cols ) = []
 | rowlistprod ( row::rows, cols) =
              rowprod(row, cols) :: rowlistprod( rows, cols );

fun matprod ( Arows, Brows ) =
              rowlistprod( Arows, transp Brows );
```

On the other hand, it is fairly easy to parallelize computations because the structure of the programs reveals the computationally independent parts which can be executed in parallel. Indeed research has shown promising results with automatic parallelization of functional programming codes [21].

4.1.1 Object Oriented Programming

The procedural languages encourage the programmer to collect algorithms and functions into libraries. However, the actual data is still left "floating" around at the mercy of the user of the libraries. Let us illustrate with an example. Let **a** and **b** denote two vectors, where **b** is a list of indices such that a_{b_i}, $i = 1 \dots n$ is a sorted list in decreasing order. A library could include algorithms to generate **b** from **a** and other functions related to manipulating the vectors. However, the user has direct access to the vectors and if not careful he might swap two elements of **a** and the sorting stored in **b** might not be correct anymore. We say that the data are in an invalid state.

The object oriented programming collects data and the interface to the algorithms for the data into one unit called a *class*. That is, not only the code but also the actual data is contained in a object. Access to data is restricted going through the interface. If the interface is well defined it should be easier to decompose the problems into self contained entities that interact in a predefined way. Due to the indirect access it is easy to avoid invalid data. In the example the "sorted list" class would need to define a swap interface that swapped the elements in both **a** and **b**.

Classes can "inherit" from other similar classes borrowing and extending the interface and functionality. In this way OO programming languages encourage reuse of code. An example could be a simple list class and an extended list class with fast access to the largest element. The specialized list could be implemented as two vectors as described in the example above and requesting the maximum value would be simple lookup of a_{b_1}. They would share the lookup of element 5 of **a**, but as described above the extended class should redefine how a swap of two elements is performed to keep the, now internal, data valid.

The object oriented ideas and techniques can be used in all languages by adhering to strict rules but actual OO languages provide constructs that simplify the implementation.

The language Simula [11] was the first language to introduce classes as an actual language construct. It didn't achieve widespread use, but the object oriented idea did. Today, we have C++ [121] (OO extension to C), Java [52] (also an OO extension to C) and Delphi (OO extension to Pascal) just to name a few of the popular OO languages. Also the newer revisions of Fortran [96] have OO features in the language that are used by of LAPACK95, see [3]. Among functional type languages OCaml [111] is a Standard ML derived language with objects. Finally, object oriented facilities were added to Matlab in version 5. For an overview of projects using object oriented techniques for numerical computations see the web-page www.oonumerics.org.

Probably the first software package for regularization of ill-posed problems was written in Simula by Eldén [34].

See also the web-page `http://www.oonumerics.org` for an overview of software and projects that use OO techniques for scientific computing. From this page it seems that 90% of object oriented mathematics are done in C++.

Before discussing object oriented programming and Matlab we will present the very basic terminology used in the following. The type of an object is called a class. An instance of a class is called an object. For example, we could say that integer is a class while the integer 3 is an instance of an integer, that is, 3 is an object. A function belonging to a class is called a method. A class defines a number of fields which are to hold the data of the object. For example the integer class could have a field called value where an instance/object would store any data, for example, the value 3. Usually it is only possible to change fields of an object through the interface.

4.2 Object Oriented Matlab

We have chosen Matlab to implement the toolbox, but why Matlab and not, for example, the more widespread C++ language? The most popular object oriented languages are C++ and Java as seen on www.oonumerics.org. However, Matlab has also been used. Examples are "DiffMann" [38] and FEMLAB [27].

The advantages of using Matlab are:

- Matlab provides a rich and powerful library of well-tested and optimized numerical algorithms.

- Matlab is an interpreted "rapid application development" environment available for multiple platforms. Interactive experiments with algorithms and data are easy as recompiling after small changes is not necessary. The same code can be used on all platforms where Matlab is available. To a large extent even the C and Fortran Mex-files are platform independent.

- Advanced integrated graphical features in Matlab makes plotting and illustration of data easy.
- Matlab is widely used in teaching and research. The previous Regularization Tools package [68] is also implemented in Matlab and M$\mathcal{OO}$Re Tools is to a large extent based upon this previous work.
- Matlab takes care of memory management and many pitfalls and programming errors are thereby avoided.

However, the many advantages come with a price. Compared to other OO languages, such as C++, we note the following disadvantages:

- As an interpreted language some overhead must be expected. Recent versions of Matlab (from version 6.5) introduce a "Just In Time" compiler that improves performance. However, some performance is still lost.
- Matlab does not allow fine-grained control of pointers and memory. Matlab does try to limit memory consumption by delaying memory allocation and copying if possible.
- The OO features of Matlab seem added as an afterthought. For example, it is not possible to force a subclass to implement an interface and thereby prevent the creation of objects that do not implement the methods that are required.

Matlab was originally constructed as a procedural programming language—in its first incarnation it was a kind of Fortran interpreter. However, as popularity grew object oriented features were added from version 5. Matlab provides the following object oriented features:

Constructors. The constructor is a special method, that is, a function, that creates and sets the data within the object to an initial (valid) state.

Encapsulation. Data, that is, the fields within the object, are not directly accessible from outside the methods of the class. This makes it possible to keep the fields in a valid state. Methods must be written to access and change data. That is, all fields are "private" in the terminology of Java and C++.

Inheritance. Classes can inherit and extend functionality from one or more other classes and thereby reuse already tested and functional code. When we say that Matrix inherits from its parent LinearOperator it is implied that Matrix is a LinearOperator. Also, LinearOperator is called a parent of Matrix and conversely Matrix is a child of LinearOperator.

Overloading. It is possible to overload functions and operators. Overloading is to redefine, for example, the multiplication operator `*` for each class. Depending on the class of the argument(s) (the operator `*` has two) the correct overloaded method is evaluated.

Polymorphism. Algorithms can be written without knowing the exact type of the used objects. The algorithm can, for example, assume that the object is a linear operator and use the interface of a linear operator. The algorithm would then work with all classes that inherit from the linear operator as these classes also have implemented the linear operator interface either by inheritance or overloading.

Notation

Because we are going to use inheritance, overloading and the other object oriented features described above we end up with several methods with the same name. When it is obvious from the context which class a method belongs to we use `method` to designate the method. In cases where it is not clear we add the class name like `classname/method`.

The next chapter goes into more detail about the choices made in the actual implementation of the M$\mathcal{OO}$Re Tools Matlab toolbox. More details on usage, syntax, directory structure and examples are placed in the toolbox tutorial in App. A.

4.3 Regularization and Objects

We have chosen not to create objects/classes for actual algorithms but to place them as methods in the base classes. That is, the OO techniques are only applied to the operators and vectors of the problems.

In this section we will go through a number of linear algebra and regularization scenarios to determine the basic classes needed for the package. The classes are divided into three categories:

Base classes. The two abstract classes LinearOperator and Vector form the top of two hierarchies of inheritance. Furthermore, we use the two classes SVDOperator and QROperator to indicate whether decompositions like the SVD and QR are possible with the particular class. All other classes inherit from one or more of these classes. The LinearOperator class holds the regularization routines based on multiplications, that is, the iterative methods. The SVDOperator class holds those routines that operate with the SVD such as the TSVD algorithm.

Utility classes. Classes that are useful in the formulation of regularization problems and algorithms. For example, we often face “block” operators, products of operators and occasionally sums of operators. That is, the utility classes are not specific to the given inverse problem.

Other classes. Classes that are specific to a specific problem and not of general use.

The first two categories are covered in detail in the following. The third category includes classes for test problems including a class Matrix which is an OO representation of the usual matrix.

4.3.1 Base Classes

To see which objects we need, we first take a look at our linear ill-posed problem

$$\mathbf{K}\mathbf{x} = \mathbf{y}, \tag{4.1}$$

where $\mathbf{x}$, $\mathbf{y}$ are vectors and $\mathbf{K}$ is a linear operator mapping a vector $\mathbf{x}$ to $\mathbf{y}$. We see that we have two basic types entities or objects:

- A linear operator $\mathbf{K}$ that can take a vector as input and output another vector possibly of another type.
- Vectors $\mathbf{x}$ and $\mathbf{y}$ representing some kind of data.

These two types of objects form the root of the class hierarchy as the classes LinearOperator and Vector.

The Interface of LinearOperator

The classes should present an interface for algorithms to use. The LinearOperator and Vector classes define an interface where the basic operations such as multiplication and addition are defined. However, the LinearOperator class does not implement most of the operations that the interface defines (in C++/Java terminology: the class is abstract). Classes that inherit from LinearOperator must implement the interface defined by LinearOperator as detailed later. Unfortunately it is not possible in Matlab to express that a subclass *must* implement a certain interface. The algorithms, on the other hand, only use the interface defined by LinearOperator and do not consider the implementation. Multiplication with a Vector is only partly implemented by the `mtimes` method in LinearOperator. The `mtimes` method first determines the classes of the two arguments and from that information it performs one of the following operations:

scalar and LinearOperator. A LinearOperator contains a scale field which is initialized by the constructor to 1. Multiplying a LinearOperator with a scalar then scales the scale field. This implies that a child class does not have to implement multiplication with a scalar because LinearOperator takes care of the scaling as we explain next.

LinearOperator and Vector. The LinearOperator is not fully capable of multiplication. In the case of LinearOperator multiplied onto a Vector the following steps take place:

1. The `sub_applytovector` method is called with the LinearOperator and Vector as arguments. If the actual class of the given LinearOperator object is, for example, the Matrix class, Matlab will execute the method `sub_applytovector` of the child class Matrix. Similarly, Matlab will execute the `sub_applytovector` method for all other child classes of LinearOperator. The `sub_applytovector` should not consider the scale field.

2. The result of the previous step, an object of class Vector (or a child thereof), is returned from `sub_applytovector`. This Vector is then scaled with the scale field of the LinearOperator and returned as the result of `mtimes`.

It is assumed that all classes inheriting from LinearOperator are capable of multiplying itself with a Vector of the proper type via the `sub_applytovector` method.

LinearOperator and LinearOperator. The procedure is somewhat similar to that of multiplication with a Vector. The steps are as follows

1. The `sub_operatormtimes` method is called with the two LinearOperator objects. Again this method should be a member of one of the actual classes given. For example, if the two objects are in fact Matrix objects, the `sub_operatormtimes` method of Matrix is called. It creates a new Matrix object containing the product which is returned.

   However, some classes do not implement `sub_operatormtimes`. In this case LinearOperator's own `sub_operatormtimes` method is called, that creates an OperatorProduct object, described later.
2. The returned LinearOperator is then scaled by the scale field of each of the LinearOperator used in the product.

In addition to overloading the `mtimes/*` method LinearOperator also assists the implementation of subclasses by implementing the methods:

`ctranspose`. Overloads the complex transpose operator, that is, writing `K'` in Matlab calls `ctranspose(K)`. `LinearOperator/ctranspose` flips the transposed flag of the operator. The flag can be read by the `gettransposed` method.

`gettransposed`. Determines whether the operator is transposed or not. This method should be used by, for example, `sub_applytovector` to decide whether it should do a multiplication with a transposed or a non-transposed operator.

`getmatrix`. Depends on the `sub_getmatrix` methods to generate a representation of the operator as the usual double array. When the child method returns, `LinearOperator/getmatrix` then scales the elements and transposes it if the transpose flag is set.

`diag`. Similar to `getmatrix`, but calls `sub_diag`. The result is scaled by the scaling factor.

`size`. Matlab uses the `size` method whenever the size of the operators is displayed by, for example, the Matlab function `whos`. It is a tedious job to get the right output for the given input if Matlab's conventions are to be followed. A subclass can instead implement `sub_size` that returns the number of columns and rows in the *non*-transposed operator. The `size` method calls the `sub_size` method and swaps the row and column count if the operator is transposed and sets up the requested output.

Note how methods named `sub_something` are used by one of the base classes.

If the operator supports creation of the SVD it inherits from SVDOperator that defines the interface to the SVD and implements a number of regularization routines that either need the SVD (such as the TSVD) or exploit the SVD for faster computations (such as Tikhonov). The SVDOperator assists by implementing the method:

`svd`. The `svd` method calls the `sub_svd` method. The `sub_svd` should compute the SVD without considering scaling of the object or whether it has been transposed. The `svd` method prepares the proper output by taking care of scaling by multiplying the singular values and taking care of whether the operator is transposed by swapping left and right singular values.

Similarly we inherit from the QROperator if the operator can compute its QR factorization. The QROperator implements:

`qr`. The `qr` method calls the `sub_qr` method which should check whether the object is transposed or not. `qr` takes care of scaling by multiplying the $\mathbf{R}$ factor.

The simple wrapper class Matrix implements all the factorizations and therefore it inherits from both LinearOperator, SVDOperator and QROperator. Thus it implements all the above mentioned `sub_op` methods. However, in some cases only the multiplication is implemented and the operator only inherits from LinearOperator. An example is the Interpolate class that implements a bilinear interpolation from a regular grid to an irregular grid via a mex-function (a function written in C or Fortran with a Matlab interface). The corresponding matrix is never explicitly constructed; see also App. D.2.

The Interface of Vector

The interface of a Vector defines multiplication with scalars, norms and inner products. It also serves as the wrapper of the ordinary vector and is therefore not an abstract class like the LinearOperator class.

Furthermore, the Vector class provides helpers to support implementation of other vector types. Most of the operations such as plus in Vector are implemented via the methods `getdata` and `setdata`. For instance, the `plus` method is implemented as:

```
function z = plus(x,y)
z = setdata( x, getdata( x ) + getdata( y ) )
```

The `Vector2D/setdata` method sets the data field of the Vector2D object and the `Vector2D/getdata` method gets the 2D array stored in its data field. The Vector2D and Vector3D classes support a large number of operations, such as plus, minus, elementwise equal etc., by implementing a `getdata/setdata` pair and relying on the implementation of Vector.

4.3.2 Utility Classes

In addition to the base classes LinearOperator, SVDOperator, QROperator and Vector a number of utility classes have proven useful in the development and implementation of algorithms for M$\mathcal{OO}$Re Tools.

Block Operators and Vectors

We often see block matrices in the formulation of algorithms. For example, we have the Tikhonov regularization problem in its least squares formulation

$$\min \left\| \begin{bmatrix} \mathbf{K} \\ \lambda \mathbf{L} \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} \right\|_2, \tag{4.2}$$

where $\mathbf{K}$ and $\mathbf{L}$ are linear operators and $\mathbf{y}$ and $\mathbf{0}$ are vectors, that is, we have

1. a linear operator composed of two linear operators "stacked" on top of each other,
2. and a vector composed of two vectors "stacked" on top of each other.

Another structure that appears often in optimization problems is the augmented 2×2 block matrix

$$\begin{bmatrix} \mathbf{D} & \mathbf{K} \\ \mathbf{K}^T & \mathbf{0} \end{bmatrix}. \tag{4.3}$$

These two examples lead us to introduce the OperatorArray and VectorStack classes. They contain a number of objects derived from the LinearOperator and Vector classes respectively and their implementation of operations are dependent on the implementation of the sub-operators and sub-vectors (the "Chain of Responsibility" pattern [41]). The two classes "go together", that is an OperatorArray may be multiplied onto a VectorStack. It is possible to nest OperatorArrays and VectorStacks to get, for example,

$$\begin{bmatrix} \mathbf{K} \\ \lambda \begin{bmatrix} \mathbf{I} \otimes \mathbf{L}_1 \\ \mathbf{L}_1 \otimes \mathbf{I} \end{bmatrix} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \mathbf{b} \\ \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \end{bmatrix}.$$

Multiplication of the above operator (transposed) and vector is computed step by step via

$$\begin{aligned} \begin{bmatrix} \mathbf{K}^T \ \lambda \left[(\mathbf{I} \otimes \mathbf{L}_1)^T \ (\mathbf{L}_1 \otimes \mathbf{I})^T \right] \end{bmatrix} \begin{bmatrix} \mathbf{b} \\ \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \end{bmatrix} &= \mathbf{K}^T \mathbf{b} + \lambda \left[(\mathbf{I} \otimes \mathbf{L}_1)^T \ (\mathbf{L}_1 \otimes \mathbf{I})^T \right] \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \\ &= \mathbf{K}^T \mathbf{b} + \lambda ((\mathbf{I} \otimes \mathbf{L}_1)^T \mathbf{0} + (\mathbf{L}_1 \otimes \mathbf{I})^T \mathbf{0}) \\ &= \mathbf{K}^T \mathbf{b} + \lambda \mathbf{0} \\ &= \mathbf{K}^T \mathbf{b}. \end{aligned}$$

Product of Operators

The weighted least squares problem

$$\min \|\mathbf{W}(\mathbf{K}\mathbf{x} - \mathbf{y})\|_2 = \min \|(\mathbf{W}\mathbf{K})\mathbf{x} - \mathbf{W}\mathbf{y}\|_2$$

is also of interest. From this problem we see that a product of operators class can be useful. The term $\mathbf{W}\mathbf{y}$ does not introduce anything new as we can just apply the weighting operator $\mathbf{W}$ to $\mathbf{y}$. We cannot always apply the weighting operator to $\mathbf{K}$ or it could be undesirable for numerical reasons, but we can apply the weighting to the result of $\mathbf{K}\mathbf{x}$. An OperatorProduct object holds a number of LinearOperator objects and multiplications with vectors are done by applying each operator in turn according to

$$(\mathbf{K}_1(\cdots(\mathbf{K}_n\mathbf{v})))$$

and according to

$$(\mathbf{K}_n^T(\cdots(\mathbf{K}_1^T\mathbf{v})))$$

in case the OperatorProduct has been transposed.

Sum of Operators

Along the same lines is the operator sum sometimes used to make a symmetric approximation to an "almost" symmetric operator

$$\mathbf{K}_s = (\mathbf{K} + \mathbf{K}^T)/2, \tag{4.4}$$

indirectly used in [60]. An OperatorSum object holds a number of LinearOperator objects. In this case a multiplication is done according to

$$(\mathbf{K}_1\mathbf{v}) + \cdots + (\mathbf{K}_n\mathbf{v}),$$

and if the OperatorSum object is transposed according to

$$(\mathbf{K}_1^T\mathbf{v}) + \cdots + (\mathbf{K}_n^T\mathbf{v}).$$

In equation (4.4) we would have a scaled (with 1/2) OperatorSum object with two "sub"-operators $\mathbf{K}$ and $\mathbf{K}^T$.

Other Utility Operators

We also add the identity operator (IdentityOperator), a class for diagonal operators (DiagonalOperator) and the null operator (NullOperator) to the utility classes as they appear often, as seen in, for example, (4.2) and (4.3). The IdentityOperator is an extremely sparse representation of the identity. Not even the ones on the diagonal are stored. Optionally the size of the operator is stored; that is, an IdentityOperator can in fact be created “sizeless”. However, operations that need the size of an operator will fail and it is therefore recommended to specify the size of the IdentityOperator when it is constructed. The NullOperator is similar to an IdentityOperator multiplied by zero. The DiagonalOperator stores the diagonal as a Vector object. An operation, such as multiplication with a vector is done through the corresponding elementwise operation between the diagonal vector and the vector. Thus DiagonalOperator relies on the interface defined by the Vector class. The DiagonalOperator can be initiated with more zero rows or columns to facilitate constructs such as

$$\begin{bmatrix}\mathbf{\Sigma} & \mathbf{0}\end{bmatrix} \quad \text{and} \quad \begin{bmatrix}\mathbf{\Sigma} \\ \mathbf{0}\end{bmatrix}.$$

The class PermutationOperator stores the permutation via a list of indices. We will use the PermutationOperator in the case study to bring the singular values of a Kronecker product into sorted order.

The utility class VectorReshape is used to keep the `sub_applytovector` methods simple. Most operators can only be applied to vectors of a specific class. For example, a KroneckerProduct2D can only be applied to Vector2D objects. However, sometimes we wish to apply the KroneckerProduct2D to a Vector object and we need to transform the object. The VectorReshape converts back and forth between a Vector object and a Vector2D, Vector3D or VectorND object. Essentially, the class works as the Matlab command `reshape`, hence the name. Because the operator does not change the elements and merely reshapes the data, it is related to the identity operator, and indeed VectorReshape is created as a child of IdentityOperator. In the case study on the Kronecker product later in this chapter we will see how the VectorReshape class is used in practice.

We have decided that all usual regularization methods take the operator, the right-hand side and an options structure as arguments. This enables us to combine methods easily. However, we cannot explicitly pass an already computed SVD to e.g. the TSVD method as it is done in the previous package. We work around this issue by introducing the OperatorSVD that caches/stores the three terms of the SVD. When we ask for the SVD of an OperatorSVD we simply return the stored objects. This approach is inspired by the linear algebra package JAMA [76] for the Java language.

All classes that represent a linear operator must inherit from LinearOperator, while some inherit from QROperator and SVDOperator, see Fig. 4.2. Note how KroneckerProduct2D and KroneckerProduct3D inherit from KroneckerProduct as they are specializations of the general N-term Kronecker product. The KroneckerProduct2D is the subject of the next section. The SparseMatrix class is similar to the Matrix class but implements the solve operation, that is, the backslash operation, via sparse QR. The VectorCollection class can contain a number of Vector objects stored as “columns”. Applying a VectorCollection $\mathbf{V}$ to a Vector $\mathbf{b}$ is computed via a linear combination of the stored vectors in $\mathbf{V}$ or, if $\mathbf{V}$ is transposed, as inner products of each stored vector in $\mathbf{V}$ with the vector $\mathbf{b}$. The classes WeightedPseudoInverse and TikhPrecond are described in §5.2 and §5.3 respectively. The three test problem examples implemented in Interpolate, SteadyHeatFEM and SteadyHeatBEM are described in App. D.

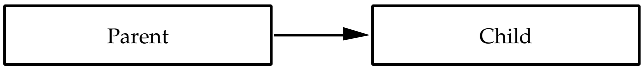


Figure 4.1: A box denotes a class. The arrow indicates that the child inherits from the Parent class. The child is also called a subclass of the parent class. It is said that the class “B” is a class “A”.

We see a similar but simpler picture for the vector types; see Fig. 4.3.

4.4 Use of Classes and Operator Overloading

We illustrate with a simple example how the object oriented abstraction, in particular the polymorphic, allows for general algorithms. The main body of a CGLS algorithm (see [13, §7.4.1]) appears as follows:

```
Kd = K*d;
alpha = normr2/(Kd'*Kd);
x  = x + alpha*d;
r  = r - alpha*Kd;
s  = K'*r;

normr2_new = s'*s;
beta = normr2_new/normr2;
normr2 = normr2_new;
d = s + beta*d;
```

As already described we can overload the basic operators in Matlab. For example, we can redefine the dyadic operator `*` via the `mtimes` method. When `K*b` is evaluated by Matlab it uses the multiplication defined by the class of the object `K`.

We observe that the operators $\mathbf{K}$ and $\mathbf{K}^T$ are applied. Hence, if the given operator class overloads multiplication and the transpose operation the code will work transparently with respect to the operator. Furthermore we see addition and subtraction of vectors, inner products and scaling of vectors by scalars, all of which a vector class must support.

Simulating OO Programming

If object oriented features such as overloading were not available the above would have to be written in another, usually less clear, form. An often used alternative is to supply the name of a function that evaluates in this case the product of the operator and a vector. In this case first line of the code would look something like

```
Kd = feval(mult, d, arg1, arg2 ...);
```

where `mult` is a function that computes `K*d` from the given arguments. A careful implementation would yield the same results as the object oriented approach. However,

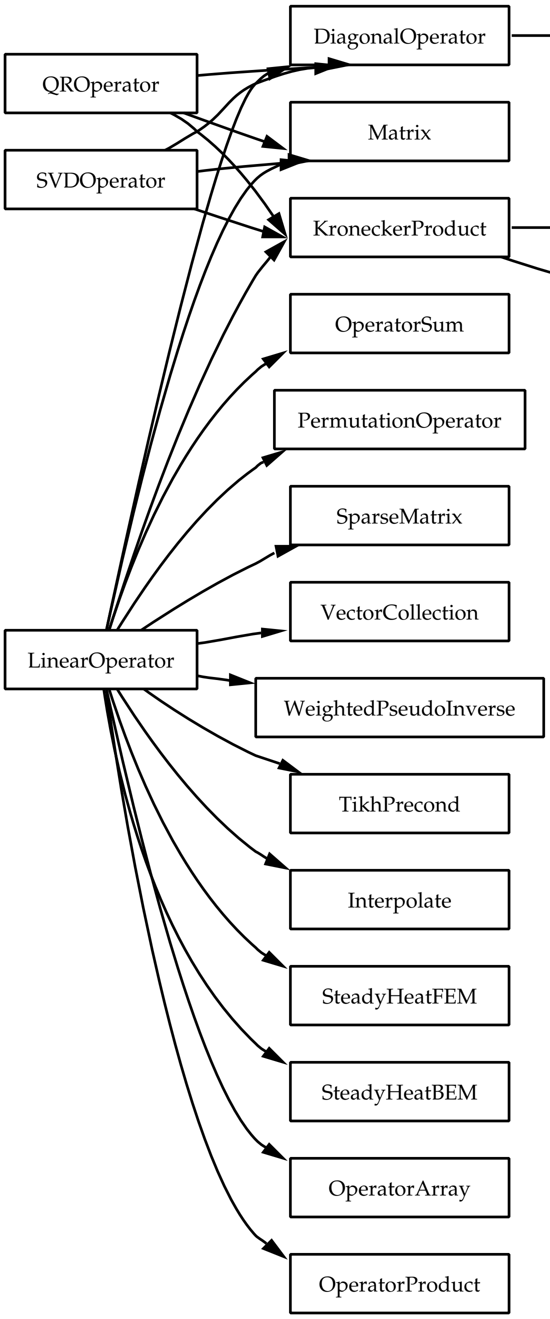


Figure 4.2: Class relationship for the LinearOperator hierarchy in M$\mathcal{OO}$Re Tools.

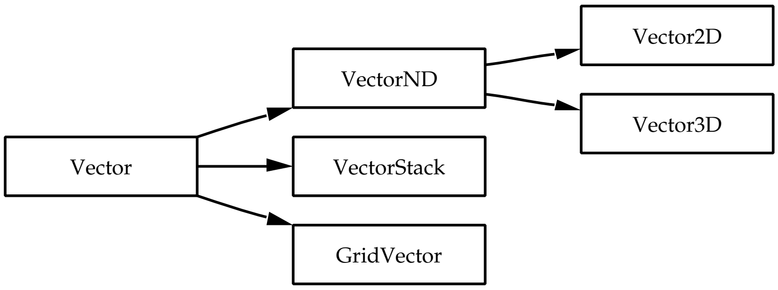


Figure 4.3: Class relationships for vector types in M$\mathcal{OO}$Re Tools.

the object oriented approach helps the programmer to keep track of which methods to use for multiplication and to hide the extra arguments.

4.5 Case Study: The Kronecker Product

In this section we will show how a two-term Kronecker product is implemented as a Matlab class KroneckerProduct2D. The extension to three and N-dimensional Kronecker products follows naturally.

4.5.1 A Kronecker Product

The Kronecker product of two matrices $\mathbf{A}$ and $\mathbf{B}$ is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2n}\mathbf{B} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}, \tag{4.5}$$

and it has among others the following very useful relations,

$$(\mathbf{A} \otimes \mathbf{B})\mathrm{vec}(\mathbf{X}) = \mathrm{vec}(\mathbf{B}\mathbf{X}\mathbf{A}^T), \tag{4.6}$$

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD}), \tag{4.7}$$

$$(\mathbf{A} \otimes \mathbf{B})^T = (\mathbf{A}^T \otimes \mathbf{B}^T), \tag{4.8}$$

where $\mathrm{vec}(\mathbf{X})$ denotes the stacking of the columns of $\mathbf{X}$ into a vector. See [92] for proofs and other relations.

The two term Kronecker product (4.5) emerges in two-dimensional problems with a separable kernel. For example, if we discretize the integral equation

$$\int_0^1 \int_0^1 k_1(x,s)k_2(y,t)f(x,y)\,dx\,dy = g(s,t),$$

we end up with a system of the type

$$(\mathbf{K}_1 \otimes \mathbf{K}_2)\text{vec}(\mathbf{F}) = \text{vec}(\mathbf{G}),$$

where $\mathbf{K}_1$ and $\mathbf{K}_2$ are discretized versions of k_1 and k_2 respectively. Using (4.6) we get

$$\mathbf{K}_2\mathbf{F}\mathbf{K}_1^T = \mathbf{G}.$$

Assuming $\mathbf{K}_1, \mathbf{K}_2 \in \mathbb{R}^{n\times n}$ the multiplication is reduced from $\mathcal{O}(n^4)$ operations to two $\mathcal{O}(n^3)$ operations. Furthermore we only need to store the two terms $\mathbf{K}_1$ and $\mathbf{K}_2$. The memory needed to store the operator is thereby reduced from one $n^2 \times n^2$ matrix to two $n \times n$ matrices. Multiplication with the transposed operator $\mathbf{K}^T$ takes advantage of (4.8) before applying (4.6).

The KroneckerProduct2D class takes advantage of these facts and stores only the two terms $\mathbf{K}_1$ and $\mathbf{K}_2$ and the multiplication in the `sub_applytovector` uses the fast multiplication rule. Thereby we have the basic requirements for a LinearOperator child in place. However, in this case we can also implement the interface required to inherit from SVDOperator.

The SVD of a Kronecker product

The Kronecker product has several useful properties. One is that most decompositions are easily created from decompositions of the terms. If we take the SVDs of $\mathbf{K}_1 = \mathbf{U}_1\mathbf{\Sigma}_1\mathbf{V}_1^T$ and $\mathbf{K}_2 = \mathbf{U}_2\mathbf{\Sigma}_2\mathbf{V}_2^T$ and use (4.7) we *almost* get an SVD

$$\mathbf{K}_1 \otimes \mathbf{K}_2 = (\mathbf{U}_1 \otimes \mathbf{U}_2)(\mathbf{\Sigma}_1 \otimes \mathbf{\Sigma}_2)(\mathbf{V}_1 \otimes \mathbf{\Sigma}_2)^T.$$

Both $(\mathbf{U}_1 \otimes \mathbf{U}_2)$ and $(\mathbf{V}_1 \otimes \mathbf{V}_2)$ are orthogonal with orthonormal columns. However, the entries in the matrix $(\mathbf{\Sigma}_1 \otimes \mathbf{\Sigma}_2)$ are not sorted and in some cases the non-zero elements are not on the diagonal (if either $\mathbf{K}_1$ or $\mathbf{K}_2$ are rectangular) which is in violation of the SVD definition, cf. Def. 2.1. Thus to fully adhere to the SVD definition we must permute $(\mathbf{\Sigma}_1 \otimes \mathbf{\Sigma}_2)$ such that the non-zero elements appear on the diagonal and are properly sorted. Let $\mathbf{P}_1$ and $\mathbf{P}_2$ denote the permutations that sort the non-zero element of each row and each column respectively, and we get by insertion

$$\mathbf{K}_1 \otimes \mathbf{K}_2 = \underbrace{(\mathbf{U}_1 \otimes \mathbf{U}_2)\mathbf{P}_1}_{\mathbf{U}}\ \underbrace{\mathbf{P}_1^T(\mathbf{\Sigma}_1 \otimes \mathbf{\Sigma}_2)\mathbf{P}_2}_{\mathbf{\Sigma}}\ \underbrace{((\mathbf{V}_1 \otimes \mathbf{\Sigma}_2)\mathbf{P}_2)}_{\mathbf{V}}{}^T.$$

Multiplication with $\mathbf{U}$ is then done by permutation followed by a multiplication with a Kronecker product.

For example, if

$$\mathbf{\Sigma}_1 = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{\Sigma}_2 = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{bmatrix},$$

we have the Kronecker product

$$\mathbf{\Sigma}_1 \otimes \mathbf{\Sigma}_2 = \begin{bmatrix} \sigma_1\sigma_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_1\sigma_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_2\sigma_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_2\sigma_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

which is neither diagonal nor sorted. Note that a vector of the (possible) non-zero elements in each row can be created from the Kronecker product

$$\begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix} \otimes \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ 0 \end{bmatrix},$$

that is, the Kronecker product of the vectors of maximum values of each row in $\mathbf{\Sigma}_1$ and $\mathbf{\Sigma}_2$ respectively. Similarly we use the maximum values of each column to create a vector of maximum values for each column of the Kronecker product. Sorting these two vectors yield the permutations needed for the SVD.

The Matlab Implementation

The implementation of the two-term Kronecker product has been achieved through the class KroneckerProduct2D. We will now go through the code for each method of the KroneckerProduct2D class. Only the comments have been removed compared to the actual contents of the files in the `KroneckerProduct2D` directory.

The class KroneckerProduct2D along with the N-term KroneckerProduct class implements:

`KroneckerProduct2D`. The constructor taking the two arguments $\mathbf{K}_1$ and $\mathbf{K}_2$ of the type LinearOperator and stores them into the `K` field of the object:

```
function Obj = KroneckerProduct2D(varargin)
switch nargin
case 0
  Obj = struct([]);
  KND = KroneckerProduct;

case 1
  if isa(varargin{1}, 'KroneckerProduct2D')
    Obj = struct([]);
    KND = varargin{1}.KroneckerProduct;
  else
    error('Incorrect input argument');
  end

case 2
  Obj = struct([]);
  KND = KroneckerProduct({varargin{1}, varargin{2}});
```

```
18
19   otherwise
20     error('Incorrect number of input arguments');
21   end
22
23   Obj = class(Obj, 'KroneckerProduct2D', KND);
24   superiorto('Vector2D');
```

In line 23 we create the object and set the generic N-term KroneckerProduct class, initialized with the two terms, as the parent of this class. That is the actual terms are stored in the parent. Line 24 says that this class is "superior" to the Vector2D class. That is, Matlab selects methods of KroneckerProduct2D before Vector2D methods. For example, if `v'*K` is called Matlab has the choice between using `mtimes` from the Vector or LinearOperator hierarchy. By default Matlab chooses the version belonging to the first argument which in this example is the vector `v`, but using `superiorto` we make sure that the multiplication is done by KroneckerProduct2D.

`sub_applytovector`. Does the fast multiply of the Kronecker product with a Vector2D object as described previously. This function is called by `mtimes` method in the parent LinearOperator class, which also takes care of scaling:

```
function y = sub_applytovector(K,x)
if isa(K, 'KroneckerProduct2D') & isa(x, 'Vector2D')
  if ~gettransposed(K)
    if size(get(K,2),2) ~= size(x,1) | ...
       size(get(K,1),2) ~= size(x,2)
      error(['Dimensions of KroneckerProduct2D terms and Vector2D do not' ...
             ' agree.']);
    end
    y = Vector2D(getterm(K,2)*getdata(x)*getterm(K,1)');
  else
    if size(get(K,2),1) ~= size(x,1) | ...
       size(get(K,1),1) ~= size(x,2)
      error(['Dimensions of KroneckerProduct2D terms and Vector2D do not' ...
             ' agree.']);
    end
    y = Vector2D(getterm(K,2)'*getdata(x)*getterm(K,1));
  end
else
  error(sprintf('Multiplication of %s with %s is not supported', ...
            class(K), class(x)));
end
```

The multiplication *could* have be taken care of by `sub_applytovector` in the parent KroneckerProduct class. But we have chosen to specialize the method for the two term Kronecker product whereby we avoid many of the details necessary in the general multiplication procedure.

The `getterm` method returns the ith term as a double array from the term stored in the parent KroneckerProduct. This is necessary because a child class has no direct access to fields of the parent. The get method also returns the ith term but as an object.

sub_getmatrix. Constructs a matrix representing the Kronecker product. This function is called getmatrix of the parent class LinearOperator that afterward adjusts scaling and whether the operator is transposed:

```
function K = sub_getmatrix(K)
K = kron(getterm(K,1),getterm(K,2));
```

The kron function is a standard Matlab routine that computes Kronecker products.

sub_operatormtimes. Performs multiplication of two Kronecker products if possible. Is like sub_applytovector called by mtimes of LinearOperator. The function checks if the multiplication is possible, that is, both terms of the product must be KroneckerProduct2D objects. Otherwise, it returns with an error and an OperatorProduct object is created by LinearOperator/mtimes. It is also required that the terms of the two KroneckerProduct2D objects have equal inner dimensions.

```
function C = sub_operatormtimes(A,B)
if isa(A, 'KroneckerProduct2D') & isa(B, 'KroneckerProduct2D')
  A1 = get(A,1); A2 = get(A,2);
  B1 = get(B,1); B2 = get(B,2);
  if gettransposed(A)
    A1 = A1';   A2 = A2';
  end
  if gettransposed(B)
    B1 = B1';   B2 = B2';
  end
  if size(A1,2) ~= size(B1,1) | ...
     size(A2,2) ~= size(B2,1),
    error('Inner dimensions of operator terms not equal.');
  end
  C = KroneckerProduct2D( A1*B1, A2*B2 );
else
  error('OPERATORPRODUCT'); % throw an OPERATORPRODUCT "exception"
end
```

sub_qr. Performs the obvious QR factorization of a Kronecker product.

$$\mathbf{K} = \mathbf{K}_1 \otimes \mathbf{K}_2 = \mathbf{QR} = (\mathbf{Q_1} \otimes \mathbf{Q_2})(\mathbf{R_1} \otimes \mathbf{R_2}).$$

Called by QROperator/qr that afterward adjusts for scaling. It is necessary for sub_qr to check if the KroneckerProduct2D object has been transposed:

```
function [Q,R] = sub_qr(K, options)
if options ~= 0
  error(sprintf('Only thin QR supported for class %s',class(K)));
end
if gettransposed(K)
```

```
  [Q1,R1] = qr(get(K,1)' ,0);
  [Q2,R2] = qr(get(K,2)' ,0);
else
  [Q1,R1] = qr(get(K,1) ,0);
  [Q2,R2] = qr(get(K,2) ,0);
end
if size(R1,1) < size(R1,2) | size(R2,1) < size(R2,2)
  error(['QR only supported for square or overdetirmed terms in', ...
         ' KroneckerProduct2D']);
end
Q = KroneckerProduct2D(Q1,Q2);
R = KroneckerProduct2D(R1,R2);
```

Note that only the thin QR factorization is supported. The triangular factor in the full QR factorization is not easily constructed with rectangular terms in the Kronecker product.

sub_solve. Solve in the least squares sense a system with a Kronecker product. Here we utilize the rule

$$(\mathbf{K}_1 \otimes \mathbf{K}_2)^\dagger = \mathbf{K}_1^\dagger \otimes \mathbf{K}_2^\dagger.$$

It is called by LinearOperator's `mrdivide` or `mldivide` methods, that also adjust for scaling:

```
function x = sub_solve(K,y)
if isa(K, 'KroneckerProduct2D') & isa(y, 'Vector2D')
  if ~gettransposed(K)
    if size(get(K,2),1) == size(y,1) & ...
       size(get(K,1),1) == size(y,2)
      x = Vector2D(getterm(K,2)  \ getdata(y) / getterm(K,1)');
    else
      error('Matrix dimensions must agree');
    end
  else
    if size(get(K,2),2) == size(y,1) & ...
       size(get(K,1),2) == size(y,2)
      x = Vector2D(getterm(K,2)' \ getdata(y) / getterm(K,1) );
    else
      error('Matrix dimensions must agree');
    end
  end
else
  error(sprintf('Solving with %s and %s not supported.', ...
               class(K), class(y)));
end
```

sub_svd. Computes the SVD of the Kronecker product as described previously. It is called by SVDOperator's `svd` method that takes care of scaling and whether the operator has been transposed:

```
function [U,S,V] = sub_svd(K,varargin)
[U1, s1, V1] = svd(get(K,1));
[U2, s2, V2] = svd(get(K,2));
```

```
4   [m1,n1] = size(get(K,1)); p1 = min(m1,n1);
5   [m2,n2] = size(get(K,2)); p2 = min(m2,n2);
6   [m,n] = size(K); p = min(m,n);
7
8   s1 = getvector(diag(s1)); s2 = getvector(diag(s2));
9
10  if nargout <= 1
11    s = kron(s1,s2);
12    if length(s) < p, s = [s; zeros( p - length(s),1)]; end
13    v = sort(s); v = flipud(v);
14    U = Vector(v);
15    return
16
17  elseif nargout == 3
18    s1r = [s1 ; zeros( max(0, m1 - p1),1)];
19    s1c = [s1 ; zeros( max(0, n1 - p1),1)];
20
21    s2r = [s2 ; zeros( max(0, m2 - p2),1)];
22    s2c = [s2 ; zeros( max(0, n2 - p2),1)];
23
24    [v, idxu] = sort(kron(s1r,s2r)); idxu = flipud(idxu);
25    PU = PermutationOperator( idxu );
26    RU = VectorReshape( m2, m1 );
27    UU = KroneckerProduct2D(U1, U2);
28
29    [v, idxv] = sort(kron(s1c, s2c)); idxv = flipud(idxv);
30    PV = PermutationOperator( idxv );
31    RV = VectorReshape( n2, n1);
32    VV = KroneckerProduct2D(V1, V2);
33
34    U = OperatorProduct( {UU, RU, PU} );
35    V = OperatorProduct( {VV, RV, PV} );
36
37    v = flipud(v); s = v(1:p);
38    if nargin == 2
39      S = DiagonalOperator(Vector( s ));
40      U = U(:,1:p);
41      V = V(:,1:p);
42    else
43      S = DiagonalOperator(Vector( s ),m,n);
44    end
45  else
46    error('Incorrect number of output arguments');
47  end
```

The `sub_svd` method calculates the SVD of the two terms (lines 2 and 3). It creates the Kronecker product of the singular values in line 11 if only the singular values are requested. Otherwise we create the singular values in lines 29 and 37. The correct permutations for the singular values are computed in lines 24 and 29. Note how we pad the vector of singular values with zeros to get the correct permutation (lines 18–22). The singular values are stored in a Vector or DiagonalOperator object. The matrices of singular vectors are created as products of three terms in OperatorProduct objects (lines 34 and

35). The products contain in addition to the Kronecker product object also the permutation as described in the previous section. The VectorReshape object is used to convert/reshape between Vector2D and Vector objects. Note that an actual Kronecker product of two matrices is never explicitly created.

`cond`. The condition number calculation is in fact *not* done by KroneckerProduct2D but by the parent KroneckerProduct. The condition number is calculated as the product of the condition number of the two terms, that is,

$$\mathrm{cond}(\mathbf{K}_1 \otimes \mathbf{K}_2) = \mathrm{cond}(\mathbf{K}_1)\mathrm{cond}(\mathbf{K}_2),$$

a fact easily seen from the discussion of the SVD. The code of the method `KroneckerProduct/cond` has the simple appearance:

```
function c = cond(K)
c = 1;
for i=1:length(K.K)
  c = c * cond( K.K{i});
end
```

where the `K` field holds the terms of the Kronecker product. Note that this KroneckerProduct has direct access to its fields whereas the child KroneckerProduct2D must go through the `get` and `getterm` methods.

`sub_size`. Also the size operation is managed by the parent KroneckerProduct class via:

```
function [m,n] = sub_size(K)
for i=1:length(K.K)
  [m(i), n(i)] = size( K.K{i} );
end
n = prod(n);  m = prod(m);
```

Observe that we could have used, that is, inherited, the more general implementations from the parent KroneckerProduct class for all operations and not just `cond` and `sub_size`. This reimplementation was done to optimize the two-term case.

The OperatorProduct object created by `sub_svd` combines three different classes; KroneckerProduct2D, PermutationOperator, and VectorReshape. Applying, for example, the first 10 columns of the matrix of singular vectors $\mathbf{U}$ to a Vector2D $\mathbf{y}$, that is, in Matlab notation

```
z = U(:,1:10)'*y
```

is done through cooperation of the objects stored in the OperatorProduct. Step by step, Matlab evaluates the product as follows:

1. The `subsref` method of OperatorProduct is called to evaluate `U(:,1:10)`. This method replaces the last term of the product, that is, the permutation $\mathbf{P}$ with the operator representing the 10 first columns of the permutation. In other

words, the OperatorProduct passes the responsibility on to the `subsref` method of PermutationOperator. We now have the operator

$$(\mathbf{U}_1 \otimes \mathbf{U}_2)\mathbf{R}\tilde{\mathbf{P}},$$

where $\tilde{\mathbf{P}}$ is composed of the first 10 columns of the original permutation.

2. The transposed flag is set and we have an object describing

$$\tilde{\mathbf{P}}^T\mathbf{R}^T(\mathbf{U}_1 \otimes \mathbf{U}_2)^T$$

3. The multiplication takes place with the recipe

$$\tilde{\mathbf{P}}^T(\mathbf{R}^T((\mathbf{U}_1 \otimes \mathbf{U}_2)^T\mathbf{y})).$$

First, Matlab uses the fast multiplication of a KroneckerProduct2D object and a Vector2D object. Secondly, the VectorReshape object reshapes the resulting Vector2D object into a Vector object, and finally the permutation picks out the correct 10 elements of the result. The result is then returned and stored in `z`.

The non-transposed operation, for example, `V(:,1:10)*z` is similar. The first step picks out the first 10 columns of the permutation stored in the OperatorProduct. The multiplication is then done with non-transposed operators. In this case the permutation creates a large Vector putting the elements of `z` into the correct positions. Then, the VectorReshape object converts the Vector into a Vector2D object and finally the Kronecker product is applied.

4.6 Algorithmic Implications

Which algorithms are suited for working with objects of the described type? Of the algorithms from the previous toolbox all, except for the LSQR and CGLS algorithms, use the SVD as the basis of computations.

Our goal is to create a toolbox capable of dealing with larger problems where creating an actual matrix and its SVD is not always a desirable or feasible option. Thus, iterative methods play a much larger role in the new toolbox compared to the previous.

We have settled for the following iterative solvers to form the basis of package:

- For square non-symmetric problems where the transpose is unavailable GMRES seems to the best choice. If the operator is symmetric MINRES can be used and if it is also positive definite we can use CG. However, in this case the range and domain of the operator need to be compatible. That is, the class of Kx must be compatible with x.

- For least squares problems LSQR or CGLS can be used.

All other problems are reduced to a linear least squares system or a series of linear least squares systems that need to be solved. However, smaller problems and problems with a special structure can still benefit from direct solvers based on factorizations. An example is the Kronecker product where the SVD is easy to compute.

A topic that often goes hand in hand with iterative methods is preconditioning. Preconditioning in the usual sense where we solve with the better conditioned and faster converging operator $\mathbf{KP}$ is easily implemented. However, when the preconditioner becomes more advanced implementation can become more complex as shown in the next section.

4.6.1 Subspace Preconditioned LSQR

The "subspace preconditioned LSQR" method presented in [71] (see App. E) describes an algorithm intended for large-scale problems.

The M$\mathcal{OO}$Re Tools package in its current state does not support a direct implementation of the algorithm and this section will describe the issues that remain to be solved. We will refer to lines in the algorithm which is listed on page 186.

The very first computation is a QR factorization of

$$\widehat{\mathbf{K}}\mathbf{V} = \begin{bmatrix} \mathbf{K} \\ \lambda \mathbf{L} \end{bmatrix} \mathbf{V} = \underbrace{\begin{bmatrix} \mathbf{Y} & \mathbf{Z} \end{bmatrix}}_{\mathbf{Q}} \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix},$$

where the columns of $\mathbf{V} \in \mathbb{R}^{m \times k}$ describe a k-dimensional subspace of the solution space. In our toolbox $\mathbf{V}$ could be represented as a collection of vectors in a VectorCollection object. Because $\widehat{\mathbf{K}}$ is an OperatorArray object the result of applying it to a Vector object is a VectorStack object. Thus, $\widehat{\mathbf{K}}\mathbf{V}$ is another VectorCollection where each "column" is a VectorStack obtained by applying $\widehat{\mathbf{K}}$ to each column of the VectorCollection $\mathbf{V}$. The problem is now to compute a QR factorization of the VectorCollection. A series of k Householder reflections seems appropriate in this case to create, column by column, the upper triangular $\mathbf{R}$.

In line 2, 4, 11, 12 and 24 the algorithm uses multiplications with $\mathbf{Y}$ and $\mathbf{Z}$. If the QR factorization of $\widehat{\mathbf{K}}\mathbf{V}$ is computed using Householder reflections we have a complete description of $\mathbf{Q}$ and thereby also of $\mathbf{Y}$ and $\mathbf{Z}$ as described in the paper.

The solution could be to implement a HouseholderCollection class that stores the appropriate Householder vectors and makes sure that the result of applying the operator results in an object of the appropriate type. In this case an operation $\mathbf{Qv}$ should return a VectorStack with the sub-vectors objects of the same type as the result of $\widehat{\mathbf{K}}\mathbf{v}$. However, this is left for future work.

4.7 Performance Implications

Introducing object oriented programming can affect performance as Matlab needs to keep track of the class, find and call the right methods etc. However, if the intended operation is time consuming compared to the extra time used for management we can neglect this factor. In this section we will briefly illustrate that the extra layer introduced by the object oriented techniques can be neglected for all but the smallest tasks and in some special cases we can use the extra layer to actually improve performance by working around quirks in Matlab.

We will look at the simple matrix-vector multiplication. Figure 4.4(a) shows timings of 300 repeated matrix-vector multiplications with and without the object oriented layer. In this setup we see that the object oriented layer costs about 0.1 seconds for 300 multiplications. When $n > 10^3$ we see that this extra cost is negligible compared to the cost of the actual matrix-vector multiplication.

On the other hand we get the possibility of working around some quirks in older versions of Matlab. Multiplication of a transposed matrix and a vector, that is,

```
>> K'*x
```

is for a large matrix $\mathbf{K}$ in some cases calculated faster by avoiding the transpose of $\mathbf{K}$,

```
>> (x'*K)'
```

This behavior is seen on Matlab 6.1 both on the command line as well as in scripts and functions. In the newer Matlab 6.5 this is only true on the command line, while the two versions are equally fast in a script and a function—probably due to the "Just In Time" compiler introduced with version 6.5. Timings using version 6.1 are shown in Fig. 4.4(b) and we see that the object oriented version is faster for problems where $n > 10^3$, because we can exploit the little trick in class Matrix's multiplication method. If the same code is executed in Matlab 6.5 we see a plot similar to Fig. 4.4(a), that is, we do not lose anything by using the trick in the newer version of Matlab. In fact computations on the command line are still accelerated with this little trick.

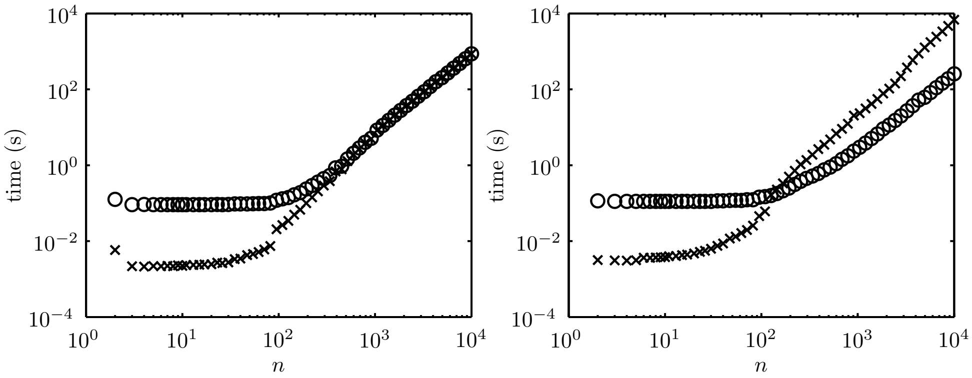
In terms of memory Matlab needs to store extra bookkeeping information about the type of an object along with the actual data for, for example, the matrix elements. The amount of extra memory needed depends on the class of the object. We have observed extra memory use ranging from 380 bytes to almost 4000 bytes for a KroneckerProdcut2D object (compared to storing the two terms).

4.8 Summary

We have introduced the object oriented programming approach with a brief justification of its advantages compared to the functional and procedural approaches.

The OO approach was used to find the entities in the linear algebra used in connection with regularization of ill-posed problems. We have demonstrated how we need just one implementation of an algorithm while the complexity of, for example. applying an operator to a vector in an iterative algorithm is kept within the operator.

In general it is not feasible to compute the SVD for large-scale problems. But in some cases we can exploit structure and still implement the SVD interface and



(a) $\mathbf{Kx}$

(b) $\mathbf{K}^T\mathbf{x}$

Figure 4.4: Timings of object oriented penalty. Timings for 300 multiplications $\mathbf{K}^T\mathbf{x}$, where $\mathbf{K} \in \mathbb{R}^{n\times n}$ with object oriented techniques ($\circ$) and without ($\times$). The computations were performed in Matlab 6.1.

use the powerful SVD based methods. The Kronecker product is an example where structure can be exploited to compute the SVD.

Finally we considered some of the negative implications of using OO programming. Some algorithms even need all new classes to be implemented before they will work. The OO approach introduces an extra layer of abstraction but we have demonstrated that the extra work is insignificant for larger problems.

Chapter 5

New Techniques

Many computational experiments have been tried since then, but little success has been reported on general linear programs. (As a general rule, negative results are rarely reported.) Stephen J. Wright, "Primal-Dual Interior-Point Methods" [128].

Mathematics is like strong alcohol. Everybody can drink it but not everybody can take it. Some get dizzy. In interview with Jean-Pierre Serre, winner of the 2003 Abel Prize, Politiken, October 24, 2003.

In this chapter we present three algorithms not found in the previous Regularization package. Several algorithms that compute upper and lower bounds of, for example, the solution norm of a Tikhonov regularized problem, are included in the new package and used in parameter choice methods for large-scale problems (see [18, 48, 49, 50]). We describe a stopping criterion added to the underlying Lanczos bidiagonalization routine that stops the bidiagonalization when the upper and lower bounds are guarantied to be close over a given interval of regularization parameters.

We also look at how the "**K**-weighted pseudo-inverse of **L**" [8, 33, 35, 101] is implemented to be more generally applicable than before. In the previous package any $\mathbf{L} \in \mathbb{R}^{p\times n}$ was assumed to have $p < n$ and to have full row rank, that is, $\mathrm{rank}(\mathbf{L}) = p$. The approach described here does not assume anything about dimensions and rank of **L**—only that the null-space of **L** is known and only intersects trivially with the null-space of **K**.

Finally we investigate a simple preconditioner for general form Tikhonov problems. We perform numerical experiments and derive an upper bound for the condition number of the preconditioned system.

In the sections on the weighted pseudo-inverse and the preconditioner we also look at the implementation details with respect to the object oriented approach.

5.1 Large-Scale Parameter Choice

If the SVD is available it is possible to compute the norms of the solutions and residuals for the standard-form Tikhonov problem fast. For example,

$$\begin{aligned}\|\mathbf{x}_\lambda\|_2 &= \|(\mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{I})^{-1}\mathbf{K}^T\mathbf{y}\|_2 \\ &= \|\mathbf{V}(\boldsymbol{\Sigma}^2 + \lambda^2\mathbf{I})^{-1}\boldsymbol{\Sigma}\mathbf{U}^T\mathbf{y}\|_2 \\ &= \|(\boldsymbol{\Sigma}^2 + \lambda^2\mathbf{I})^{-1}\boldsymbol{\Sigma}\mathbf{U}^T\mathbf{y}\|_2 \\ &= \left(\sum_{i=1}^{n}\left(\frac{\sigma_i\mathbf{u}_i^T\mathbf{y}}{\sigma_i^2+\lambda^2}\right)^2\right)^{1/2}\end{aligned}$$

shows that, when we have computed $\mathbf{U}^T\mathbf{y}$ ($\mathcal{O}(n^2)$ operations), we can compute the solution norm $\|\mathbf{x}_\lambda\|_2$ for a regularization parameter λ in just $\mathcal{O}(n)$ operations.

The parameter choice methods in the previous package all use the SVD to find the solution and residual norms needed in the particular parameter choice method. As the SVD is unavailable for most large-scale problems we seek other methods to estimate the norms we need. Work by Golub and von Matt [50] showed how upper and lower bounds on solution and residual norms could be computed through a fascinating connection between Gauss quadrature and Lanczos tridiagonalization (Alg. 1) and bidiagonalization (Alg. 3). The new development in this section is a stopping criterion that, given a range of regularization parameters, stops for the bidiagonalization algorithm when the ratio between upper and lower bounds for the residual norm is guaranteed to be smaller than a given tolerance. But first a sketch of how the bidiagonalization enters the picture.

The upper and lower bounds are a consequence of applying Gauss and Gauss-Radau quadrature (see, for example, [28]), with errors of opposite sign, to an integral related to the quadratic form

$$\mathbf{z}^T f(\mathbf{H},\lambda)\mathbf{z}, \tag{5.1}$$

where $\mathbf{z} \in \mathbb{R}^n$, $\mathbf{H} \in \mathbb{R}^{n\times n}$ is symmetric and f is an analytic function.

Let $\mathbf{H} = \mathbf{V}\mathbf{M}\mathbf{V}^T$ be an eigenvalue decomposition where the eigenvalues $\mu_1 \leq \cdots \leq \mu_n$ are sorted in non-decreasing order in the diagonal matrix $\mathbf{M}$. In the following we will use an integration interval $[a;b]$ that contains the eigenvalues, that is, $a < \mu_1$ and $\mu_n < b$. Then we can rewrite the quadratic form $\mathbf{z}^T f(\mathbf{H},\lambda)\mathbf{z}$ as an integral of the Riemann-Stieltjes type (see e.g. [28])

$$\mathbf{z}^T f(\mathbf{H},\lambda)\mathbf{z} = \mathbf{h}^T f(\mathbf{M},\lambda)\mathbf{h} = \sum_{i=1}^{n} f(\mu_i,\lambda)h_i^2 = \int_a^b f(\mu,\lambda)d\omega(\mu),$$

where $\mathbf{h} = \mathbf{V}^T\mathbf{z}$ and the measure $\omega(\mu)$ is the staircase function

$$\omega(\mu) = \sum_{\{i|\mu_i<\mu\}} h_i$$

or in other words, $\omega(\mu)$ is the sum of all eigenvalues μ_i less than μ. The integral can be approximated by quadrature rules, that is,

$$\int_a^b f(\mu,\lambda)d\omega(\mu) \approx \sum_{i=1}^{k} f(x_i,\lambda)w_i,$$

with weights $\{w_i\}$ and abscissas $\{x_i\}$. By a clever choice of two quadrature rules with errors of opposite signs we find upper and lower bounds for the integral and thus the quadratic form (5.1). Golub and von Matt [49, 50] show how the integral can be evaluated by a Gauss and a Gauss-Radau quadrature. The quadrature rules boil down to evaluating

$$\mathbf{e}_1^T f(\mathbf{T}_k,\lambda)\mathbf{e}_1 \quad \text{and} \quad \mathbf{e}_1^T f(\tilde{\mathbf{T}}_k,\lambda)\mathbf{e}_1,$$

where $\mathbf{T}_k$ is the triangular matrix from the kth step of the Lanczos algorithm (Alg. 1) on $\mathbf{H}$ with $\mathbf{z}$ as the starting vector. The matrix $\tilde{\mathbf{T}}_k$ is $\mathbf{T}_k$ modified to have a zero eigenvalue which corresponds to fixing an abscissa in the quadrature rule to zero giving a Gauss-Radau quadrature rule, see [44, 47]. But why this interest in expressions of the type (5.1)?

The solution to a standard-form Tikhonov problem,

$$\mathbf{x}_\lambda = (\mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{I})^{-1}\mathbf{K}^T\mathbf{y},$$

has the squared norm

$$\begin{aligned} \|\mathbf{x}_\lambda\|_2^2 &= \|(\mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{I})^{-1}\mathbf{K}^T\mathbf{y}\|_2^2 \\ &= (\mathbf{K}^T\mathbf{y})(\mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{I})^{-2}(\mathbf{K}^T\mathbf{y}) \\ &= (\mathbf{K}^T\mathbf{y})f(\mathbf{K}^T\mathbf{K},\lambda)(\mathbf{K}^T\mathbf{y}), \end{aligned} \tag{5.2}$$

where $f(t,\lambda) = 1/(t+\lambda^2)^2$. Moreover, using the Sherman-Morrison-Woodbury formula we can write the squared norm of the residual as

$$\begin{aligned} \|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2^2 &= \|(\mathbf{K}(\mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{I})^{-1}\mathbf{K}^T - \mathbf{I})\mathbf{y}\|_2^2 \\ &= \|\lambda^2(\mathbf{K}\mathbf{K}^T + \lambda^2\mathbf{I})^{-1}\mathbf{y}\|_2^2 \\ &= \lambda^4\mathbf{y}^T(\mathbf{K}\mathbf{K}^T + \lambda^2\mathbf{I})^{-2}\mathbf{y} \\ &= \lambda^4\mathbf{y}^T f(\mathbf{K}\mathbf{K}^T,\lambda)\mathbf{y}. \end{aligned} \tag{5.3}$$

Only the residual norm is used to determine the regularization parameter in the discrepancy principle, while both the residual and solution norms are used to plot the L-curve and visually find the corner. To compute the GCV we need not only the residual and solution norms, but also the trace of $(\mathbf{I} - \mathbf{K}(\mathbf{K}^T\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{K}^T))$. Using the Sherman-Morrison-Woodbury formula and Hutchinson's trace estimator (see [80] for details) we get the unbiased estimate

$$\begin{aligned} \mathrm{trace}(\mathbf{I} - \mathbf{K}(\mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{I})^{-1}\mathbf{K}^T) &= \mathrm{trace}(\lambda^2(\mathbf{K}\mathbf{K}^T + \lambda^2\mathbf{I})^{-1}) \\ &\approx \lambda^2\mathbf{u}^T(\mathbf{K}\mathbf{K}^T + \lambda^2\mathbf{I})^{-1}\mathbf{u} \\ &= \lambda^2\mathbf{u}^T g(\mathbf{K}\mathbf{K}^T,\lambda)\mathbf{u}, \end{aligned}$$

where $\mathbf{u}$ is a vector of zeros and ones with equal probability and $g(t,\lambda) = 1/(t+\lambda^2)$.

To actually determine a regularization parameter using, for example, the discrepancy principle, we need to find λ such that $\|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2^2 - \delta^2 = 0$, where δ is the noise-level in the right-hand side. Efficient root finding methods utilize derivatives of the solution and residual norms with respect to λ. For example, we have

$$\begin{aligned}\frac{d}{d\lambda}\|\mathbf{x}_\lambda\|_2^2 &= (\mathbf{K}^T\mathbf{y})f'(\mathbf{K}^T\mathbf{K},\lambda)(\mathbf{K}^T\mathbf{y}) \\ &= -4\lambda(\mathbf{K}^T\mathbf{y})(\mathbf{K}^T\mathbf{K}+\lambda^2\mathbf{I})^{-3}(\mathbf{K}^T\mathbf{y}),,\end{aligned} \tag{5.4}$$

where $f'(t,\lambda) = -4\lambda/(t+\lambda^2)^3$.

In the discussion of the bidiagonalization algorithms in Sec. 2.3 we learned that the lower bidiagonal matrix $\mathbf{B}_k$ from BIDIAGLOW, or to be more precise the leading $k \times k$ part,

$$\bar{\mathbf{B}}_k = \begin{bmatrix} \alpha_1 & & & \\ \beta_1 & \alpha_2 & & \\ & \ddots & \ddots & \\ & & \beta_{k-1} & \alpha_k \end{bmatrix},$$

is related to the tridiagonal matrix $\bar{\mathbf{T}}_k$ from the Lanczos tridiagonalization algorithm on $\mathbf{K}\mathbf{K}^T$ with $\mathbf{y}$ as starting vector as

$$\bar{\mathbf{B}}_k\bar{\mathbf{B}}_k^T = \bar{\mathbf{T}}_k.$$

From the QR factorization $\mathbf{B}_k = [\mathbf{Q}_1\,\mathbf{Q}_0][\mathbf{R}_k^T\,\mathbf{0}]^T$ we get an upper bidiagonal $\mathbf{R}_k \in \mathbb{R}^{k\times k}$ where

$$\mathbf{R}_k^T\mathbf{R}_k = \mathbf{T}_k.$$

where $\mathbf{T}_k$ equals the tridiagonal obtained from Lanczos tridiagonalization of $\mathbf{K}^T\mathbf{K}$ with $\mathbf{K}^T\mathbf{y}$ as starting vector. Using $\bar{\mathbf{T}}_k$ and $\mathbf{T}_k$ corresponds to a Gauss quadrature rule with the error term

$$\frac{f^{(2k)}(c,\lambda)}{(2k)!}\int_a^b \prod_{i=1}^k (t-w_i)^2 d\omega(t),$$

where $f^{(2k)}(c,\lambda)$ is the $2k$th derivative of $f(t,\lambda)$ with respect to t evaluated at c which is bounded by the integration interval $a < c < b$. The integral term is always positive. The $2k$th derivative of $f(t,\lambda)$ with respect to t is

$$f^{(2k)}(t,\lambda) = (2k!)(-1)^{2k}(t+\lambda^2)^{-2(k+1)} = (2k!)(f(t,\lambda))^{(2(k+1))} > 0$$

Because $\bar{\mathbf{T}}_k$ and $\mathbf{T}$ are positive definite we can safely set $a = 0$ implying that $c > 0$. Thus, using the Gauss quadrature gives a positive error for this particular function and we effectively have a lower bound on residual and solution norms. The Gauss-Radau quadrature rule where we fix one abscissa to zero gives a negative quadrature error for a function such as f and thereby we get an lower bound, see [50] for details. The modification to $\bar{\mathbf{T}}_k$ and $\mathbf{T}_k$ that creates a Gauss-Radau quadrature rule is obtained by removing the last row of $\bar{\mathbf{B}}_k^T$ and $\mathbf{R}_k$ respectively. For example,

$$\tilde{\bar{\mathbf{T}}}_{k-1} = \tilde{\bar{\mathbf{B}}}_k \tilde{\bar{\mathbf{B}}}_k^T \quad \text{where} \quad \tilde{\bar{\mathbf{B}}}_k^T = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ & \alpha_2 & \beta_2 & & \\ & & \ddots & \ddots & \\ & & & \alpha_{k-1} & \beta_{k-1} \end{bmatrix} \in \mathbb{R}^{(k-1)\times k}.$$

Thus, we can replace the triangular matrices with the corresponding bidiagonal "factorizations" and we get for the solution and residual norms the bounds

$$\begin{aligned} \|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2^2 &\geq \lambda^4 \mathbf{e}_1^T f(\bar{\mathbf{B}}_k \bar{\mathbf{B}}_k^T, \lambda)\mathbf{e}_1 = \mathbf{e}_1^T(\bar{\mathbf{B}}_k \bar{\mathbf{B}}_k^T + \lambda^2 \mathbf{I})^{-2}\mathbf{e}_1, \\ \|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2^2 &\leq \lambda^4 \mathbf{e}_1^T f(\tilde{\bar{\mathbf{B}}}_k \tilde{\bar{\mathbf{B}}}_k^T, \lambda)\mathbf{e}_1 = \mathbf{e}_1^T(\tilde{\bar{\mathbf{B}}}_k \tilde{\bar{\mathbf{B}}}_k^T + \lambda^2 \mathbf{I})^{-2}\mathbf{e}_1, \\ \|\mathbf{x}_\lambda\|_2^2 &\geq \quad \mathbf{e}_1^T f(\mathbf{R}_k^T \mathbf{R}_k, \lambda)\mathbf{e}_1 = \mathbf{e}_1^T(\mathbf{R}_k^T \mathbf{R}_k + \lambda^2 \mathbf{I})^{-2}\mathbf{e}_1, \\ \|\mathbf{x}_\lambda\|_2^2 &\leq \quad \mathbf{e}_1^T f(\tilde{\mathbf{R}}_k^T \tilde{\mathbf{R}}_k, \lambda)\mathbf{e}_1 = \mathbf{e}_1^T(\tilde{\mathbf{R}}_k^T \tilde{\mathbf{R}}_k + \lambda^2 \mathbf{I})^{-2}\mathbf{e}_1. \end{aligned}$$

Owing to the simple nature of f, we can compute the upper bounds on the solution and residual norms from the solutions of the following least squares systems

$$\|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2^2 \geq \lambda^4 \mathbf{z}^T\mathbf{z} \qquad \text{where } \mathbf{z} = \underset{\mathbf{z}}{\operatorname{argmin}} \left\| \begin{bmatrix} \bar{\mathbf{B}}_k^T \\ \lambda \mathbf{I} \end{bmatrix} \mathbf{z} - \begin{bmatrix} \mathbf{0} \\ \mathbf{e}_1/\lambda \end{bmatrix} \right\|_2^2 \tag{5.5}$$

$$\|\mathbf{x}_\lambda\|_2^2 \geq \mathbf{w}^T\mathbf{w} \qquad \text{where } \mathbf{w} = \underset{\mathbf{w}}{\operatorname{argmin}} \left\| \begin{bmatrix} \mathbf{R}_k \\ \lambda \mathbf{I} \end{bmatrix} \mathbf{z} - \begin{bmatrix} \mathbf{0} \\ \mathbf{e}_1/\lambda \end{bmatrix} \right\|_2^2. \tag{5.6}$$

Replacing the bidiagonal matrices $\bar{\mathbf{B}}_k^T$ and $\mathbf{R}_k$ with $\tilde{\bar{\mathbf{B}}}_k^T$ and $\tilde{\mathbf{R}}_k^T$ respectively yields least squares systems used to compute the lower bounds.

The trace estimator is computed using the solution of the least squares problem in (5.5) using

$$\operatorname{trace}(\lambda^2(\mathbf{K}\mathbf{K}^T + \lambda^2\mathbf{I})^{-1}) \leq \lambda^2 \mathbf{e}_1^T \mathbf{z} \text{ where } \mathbf{z} = \underset{\mathbf{z}}{\operatorname{argmin}} \left\| \begin{bmatrix} \bar{\mathbf{B}}_k^T \\ \lambda \mathbf{I} \end{bmatrix} \mathbf{z} - \begin{bmatrix} \mathbf{0} \\ \mathbf{e}_1/\lambda \end{bmatrix} \right\|_2^2, \tag{5.7}$$

where the Gauss quadrature formula computes an upper bound as $g^{(2k)}(t) < 0$ for $a < t < b$.

The derivative example in (5.4) is computed by

$$\frac{d}{d\lambda}\|\mathbf{x}_\lambda\|_2^2 \leq \mathbf{v}^T\mathbf{w} \quad \text{where} \quad \mathbf{v} = \underset{\mathbf{v}}{\operatorname{argmin}} \left\| \begin{bmatrix} \mathbf{R}_k \\ \lambda \mathbf{I} \end{bmatrix} \mathbf{v} - \begin{bmatrix} \mathbf{0} \\ \mathbf{w}/\lambda \end{bmatrix} \right\|_2,$$

where $\mathbf{w}$ is the solution from the least squares problem in (5.6).

We will now show how to iteratively compute the upper and lower bounds of the residual norm. That is, we will monitor the norm of the least squares solution in (5.5) with and without setting $\alpha_k = 0$. Setting $\alpha_k = 0$ in the least squares problem is equivalent to removing the last row of $\bar{\mathbf{B}}_k$. Because the function $f(\mu, \lambda)$ has a singularity at $\lambda = 0$ the quadrature rules are less accurate for λ near zero and the difference between upper and lower bounds will be largest near zero. Thus, we only monitor the upper and lower bounds for the smallest regularization parameter we are considering. Hanke [58] recently showed that the upper and lower bounds improve monotonically with the number of Lanczos bidiagonalization steps. That is, when the stopping criterion has turned true it will not become false in a later iteration.

Figure 5.1 shows the effect of applying a sequence of Givens rotations to a system of the type (5.5). The sequence computes an upper bidiagonal matrix. The problem is now reduced to the least squares problem

$$\mathbf{z} = \underset{\mathbf{z}}{\operatorname{argmin}} \left\| \begin{bmatrix} \mathbf{U} \\ \mathbf{0} \end{bmatrix} \mathbf{z} - \begin{bmatrix} \boldsymbol{\phi} \\ \psi \end{bmatrix} \right\|_2,$$

where $\mathbf{U}$ is an upper bidiagonal matrix.

Computing $\mathbf{z} = \mathbf{U}^{-1}\boldsymbol{\phi}$ is a matter of a simple back-substitution, but all elements of $\mathbf{z}$ change at each iteration and all elements of $\mathbf{U}$ need to be stored. Instead we use another sequence of Givens rotations to compute a lower bidiagonal matrix. The sequence is illustrated in Fig. 5.2. Writing the resulting lower bidiagonal matrix $\mathbf{L} = \mathbf{U}\mathbf{Q}^T$ and $\mathbf{w} = \mathbf{Q}\mathbf{z}$, where $\mathbf{Q} = \mathbf{G_k} \cdots \mathbf{G_1}$ represents the sequence of Givens rotations, we have rewritten the problem to

$$\underset{\mathbf{z}}{\operatorname{argmin}} \|\mathbf{U}\mathbf{Q}^T\mathbf{Q}\mathbf{z} - \boldsymbol{\phi}\|_2 = \|\mathbf{L}\mathbf{w} - \boldsymbol{\phi}\|_2. \tag{5.8}$$

Because we only need the norm of $\mathbf{z}$ (and not $\mathbf{z}$ itself) and $\|\mathbf{w}\|_2^2 = \|\mathbf{Q}\mathbf{z}\|_2^2 = \|\mathbf{z}\|_2^2$ we find our result by monitoring the norm of $\mathbf{w}$ and not the actual solution. Note that the leading $(k-1) \times (k-1)$ part of $\mathbf{L}$ does not change after iteration k and as a consequence neither does the leading part of $\mathbf{w}$. When we evaluate the Gauss-Radau quadrature rule we have $\alpha_k = 0$. The leading part of $\mathbf{L}$ and therefore also $\mathbf{W}$ are independent of α_k and the computation of the leading part of $\mathbf{w}$ can be used by both the upper and lower bound computation. When we estimate with $\alpha_k = 0$ we use $\bar{\lambda}_k$ as ρ_k and $\bar{\psi}_k$ as ϕ_k. The algorithm, created from these observations, is listed in Alg. 5. We suggest using the stopping criterion

$$\frac{\log_{10}(\tilde{\eta}/\eta)}{2} < \tau, \tag{5.9}$$

where τ is some tolerance and $\tilde{\eta}$ and η are the upper and lower bounds respectively. The criterion insures that the upper and lower bounds on the residual norm are within a factor of 10^τ of each other. The left-hand side is divided by two because η and $\tilde{\eta}$ approximate the norms squared.

$$
\begin{matrix} \to \\ \\ \\ \to \\ \\ \\ \end{matrix}
\begin{bmatrix}
\alpha_1 & \beta_1 & & \\
 & \alpha_2 & \beta_2 & \\
 & & \alpha_3 & \\
\lambda & & & 1/\lambda \\
 & \lambda & & \\
 & & \lambda &
\end{bmatrix}
\xrightarrow{3\text{–}5}
\begin{matrix} \\ \\ \\ \to \\ \to \\ \\ \end{matrix}
\begin{bmatrix}
\rho_1 & \xi_1 & & \phi_1 \\
 & \alpha_2 & \beta_2 & \\
 & & \alpha_3 & \\
0 & \nu_1 & & \hat{\psi}_1 \\
 & \lambda & & \\
 & & \lambda &
\end{bmatrix}
\xrightarrow{12\text{–}13}
$$

$$
\begin{matrix} \\ \to \\ \\ \\ \to \\ \\ \end{matrix}
\begin{bmatrix}
\rho_1 & \xi_1 & & \phi_1 \\
 & \alpha_2 & \beta_2 & \\
 & & \alpha_3 & \\
 & 0 & & \psi_1 \\
 & \bar{\lambda}_1 & & \bar{\psi}_2 \\
 & & \lambda &
\end{bmatrix}
\xrightarrow{14\text{–}16}
\begin{matrix} \\ \\ \\ \\ \to \\ \to \end{matrix}
\begin{bmatrix}
\rho_1 & \xi_1 & & \phi_1 \\
 & \rho_2 & \xi_2 & \phi_2 \\
 & & \alpha_3 & \\
 & & & \psi_1 \\
 & 0 & \nu_2 & \hat{\psi}_2 \\
 & & \lambda &
\end{bmatrix}
\xrightarrow{12\text{–}13}
$$

$$
\begin{matrix} \\ \\ \to \\ \\ \\ \to \end{matrix}
\begin{bmatrix}
\rho_1 & \xi_1 & & \phi_1 \\
 & \rho_2 & \xi_2 & \phi_2 \\
 & & \alpha_3 & \\
 & & & \psi_1 \\
 & & 0 & \psi_2 \\
 & & \bar{\lambda}_3 & \hat{\psi}_3
\end{bmatrix}
\xrightarrow{14\text{–}16}
\begin{bmatrix}
\rho_1 & \xi_1 & & \phi_1 \\
 & \rho_2 & \xi_2 & \phi_2 \\
 & & \rho_3 & \phi_3 \\
 & & & \psi_1 \\
 & & 0 & \psi_2 \\
 & & 0 & \psi_3
\end{bmatrix}
$$

Figure 5.1: Givens rotations to compute upper bidiagonal. Arrows show the rows affected by the following Givens rotation. The last column of each matrix shows the result of applying the Givens rotations to the right-hand side. The numbers above the arrows indicate the corresponding lines in Alg. 5.

That is, the bidiagonalization is stopped when we are certain that the upper and lower bounds on the residual norm are closer than a given tolerance for the smallest regularization parameter considered. The bidiagonal matrix can now be used to estimate upper and lower bounds for residual and solution norms to form L-curve ribbons and with the derivatives we can also estimate upper and lower bounds for the L-curve curvature [18]. A nice overview of different applications can be found in [50].

$$\begin{array}{c} \begin{array}{ccc} \downarrow & \downarrow & \end{array} \\ \begin{bmatrix} \rho_1 & \xi_1 & \\ & \rho_2 & \xi_2 \\ & & \rho_3 \end{bmatrix} \end{array} \xrightarrow{1} \begin{array}{c} \begin{array}{ccc} & \downarrow & \downarrow \end{array} \\ \begin{bmatrix} \gamma_1 & 0 & \\ \theta_1 & \bar{\rho}_2 & \xi_2 \\ & & \rho_3 \end{bmatrix} \end{array} \xrightarrow{2} \begin{array}{c} \\ \begin{bmatrix} \gamma_1 & & \\ \theta_1 & \gamma_2 & 0 \\ & \theta_2 & \bar{\rho}_3 \end{bmatrix} \end{array}$$

Figure 5.2: Givens rotations to compute lower bidiagonal corresponding to lines 18–20 of Alg. 5. Affected columns are indicated with arrows.

Numerical Experiments

We will now illustrate how lowering the tolerance in the stopping criteria affects Regińska's L-curve parameter choice method. Regińska's L-curve parameter choice method finds the parameter from

$$\lambda = \underset{\lambda}{\operatorname{argmin}}\{\|\mathbf{K}\mathbf{x}_\lambda - \mathbf{y}\|_2 \|\mathbf{x}_\lambda\|_2\},$$

where $\mathbf{x}_\lambda$ is the Tikhonov regularized solution with regularization parameter λ. The lower bounds of the product are easily found by multiplying the lower bounds of the residual and solution norms. Likewise, we find upper bounds by multiplying upper bounds on residual and solution norms.

After just 6 bidiagonalization steps we see in Fig. 5.3(a) that the minimum of the upper bound is a good approximation of the regularization parameter. On the other hand we see that the bounds are wide apart for $\lambda = 10^{-4}$. Decreasing the tolerance to $\tau = 0.05$ yields after 24 iterations of the bidiagonalization algorithm the bounds displayed in Fig. 5.3(b). Now both lower and upper bounds agree on the minimum. In this example Regińska's L-curve chooses a regularization parameter that undersmooths.

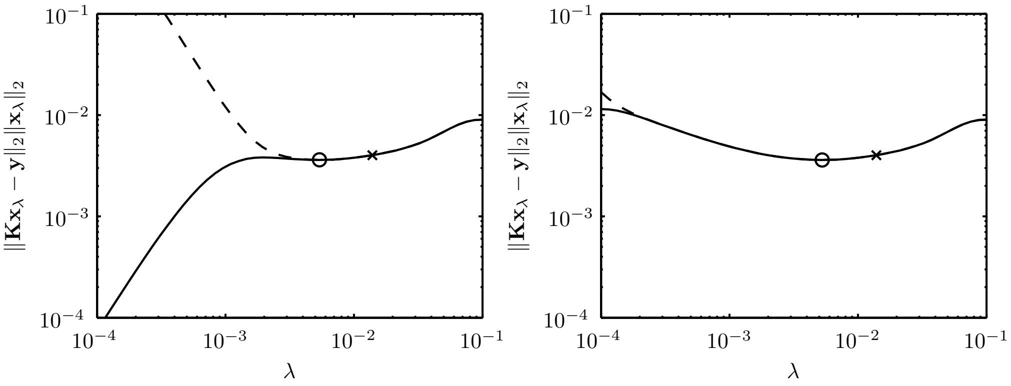
The above example seems very favorable to the bounding methods as only 16 iterations are needed to compute tight bounds. However, using a bounding method with the BLUR test problem yields much slower convergence, see Fig. 5.4. The convergence can be accelerated in terms of iterations by using reorthogonalization of the Lanczos vectors in the bidiagonalization but at the expense of more work. Also for this test case Regińska's L-curve finds an undersmoothing regularization parameter.

Algorithm 5: Lanczos Bidiagonalization II

$[\mathbf{U},\mathbf{V},\mathbf{B}] = \textsc{bidiag}(\mathbf{K},\mathbf{y},\ k,\ \tau)$

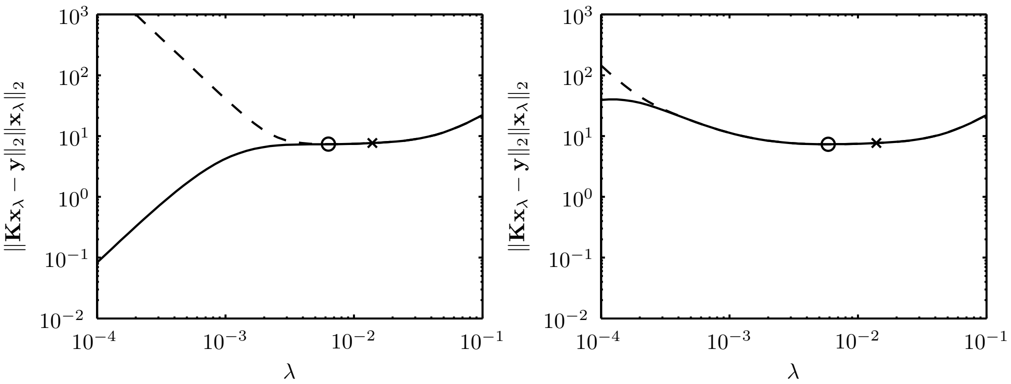
Extra lines for Alg. 3 to compute upper and lower bounds. The first iteration, that is, the computation of β_1 and α_1 is moved outside the loop to facilitate initialization of recursion formulas. Lines 11–27 are inserted between lines 6 and 7 of Alg. 3. See also Fig. 5.1 and 5.2.

1: Compute α_1 % Lines 5–6 in Alg. 3
2: $\bar{\psi}_1 = 1/\lambda$
3: $[n_1, c_1, s_1] \leftarrow \textsc{givens}(\alpha_1, \lambda)$
4: $\rho_1 = n_1; \quad \phi_1 = s_1\bar{\psi}_1$
5: $\hat{\psi}_1 = c_1\bar{\psi}_1$
6: $i \leftarrow 2; \quad c_2 \leftarrow 1; \quad \theta_0 = 0; \quad z_0 = 0; \quad \eta_z \leftarrow 0$
7: $\eta = (\phi_1/\rho_1)^2; \quad \tilde{\eta} = (1/\lambda)^4$
8: Compute β_1 % Lines 7–8 in Alg. 3
9: **while** $\log_{10}(\tilde{\eta}/\eta) > 2\tau$ and $i < k$ **do**
10: Compute α_i % Lines 5–6 in Alg. 3
11: $\xi_{i-1} = c_1\beta_{i-1}; \quad \nu_i = -s_1\beta_{i-1}$
12: $[n_0, c_0, s_0] \leftarrow \textsc{givens}(\lambda, \nu_i)$
13: $\bar{\lambda} = n_0; \quad \bar{\psi}_i = s_0\hat{\psi}_{i-1}$
14: $[n_1, c_1, s_1] \leftarrow \textsc{givens}(\alpha_i, \bar{\lambda})$
15: $\rho_i = n_1; \quad \phi_i = s_1\bar{\psi}_i$
16: $\hat{\psi}_i = c_1\bar{\psi}_i$
17: $\bar{\rho}_{i-1} = c_2\rho_{i-1}$
18: $[n_2, c_2, s_2] \leftarrow \textsc{givens}(\bar{\rho}_{i-1}, \xi_{i-1})$
19: $\gamma_{i-1} = n_2$
20: $\theta_{i-1} = s_2\rho_i$
21: $z_{i-1} = (\phi_{i-1} - \theta_{i-2}z_{i-2})/\gamma_{i-1}$ % New fixed element
22: $\eta_z \leftarrow \eta_z + z_{i-1}^2$ % Update norm of fixed elements
23: $z = (\phi_i - \theta_{i-1}z_{i-1})/(c_2\rho_i)$ % Current last element
24: $\eta \leftarrow \eta_z + z^2$ % Add to fixed norm
25: $\tilde{\theta}_{i-1} = s_2\bar{\lambda}$ % Assume $\alpha_i = 0$
26: $\tilde{z} = (\bar{\psi}_i - \tilde{\theta}_{i-1}z_{i-1})/(c_2\bar{\lambda})$ % Current last element with $\alpha = 0$
27: $\tilde{\eta} \leftarrow \eta_z + \tilde{z}^2$ % Add to fixed
28: Compute β_i % Lines 7–8 in Alg. 3
29: $i \leftarrow i + 1$
30: **end while**
31: Create matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{B}$ as in Alg. 3



(a) Tolerance $\tau = 2$ (6 iterations) (b) Tolerance $\tau = 0.05$ (24 iterations)

Figure 5.3: Bounds for Regińska's L-curve. The test problem is DERIV2 with $\mathbf{K} \in \mathbb{R}^{1024\times 1024}$ where a vector of normal distributed noise with norm $\|\mathbf{e}\|_2 = 10^{-2}\|\mathbf{y}\|_2$ was added to the right-hand side. The circles ($\circ$) indicate the minimum of the upper bound curve while the optimal regularization parameter w.r.t. the true solution is marked with ($\times$). Bidiagonalizations are computed without reorthogonalization of Lanczos vectors.



(a) Tolerance $\tau = 2$ (203 iterations) (b) Tolerance $\tau = 0.1$ (2333 iterations)

Figure 5.4: Bounds for Regińska's L-curve with the BLUR test problem, with $\sigma_x = \sigma_y = 1.5$, $\mathbf{K} \in \mathbb{R}^{1024\times 1024}$ and normal distributed noise with norm $\|\mathbf{e}\|_2 = 10^{-2}\|\mathbf{y}\|_2$. The optimal regularization parameter using Regińska's L-curve is marked with (o), while the optimal parameter w.r.t. the solution is marked (x). Note that the bidiagonalizations were computed *without* reorthogonalization of the Lanczos vectors.

5.2 The K-weighted Pseudo-Inverse

The **K**-weighted pseudo-inverse is used to transform a general-form Tikhonov problem

$$\min \left\{ \|\mathbf{Kx} - \mathbf{y}\|_2^2 + \lambda^2 \|\mathbf{Lx}\|_2^2 \right\}, \tag{5.10}$$

into a standard-form Tikhonov problem

$$\min \left\{ \|\bar{\mathbf{K}}\bar{\mathbf{x}} - \bar{\mathbf{y}}\|_2^2 + \lambda^2 \|\bar{\mathbf{x}}\|_2^2 \right\}. \tag{5.11}$$

If $\mathbf{L}$ is square and invertible, we simply use the transformations $\bar{\mathbf{K}} = \mathbf{KL}^{-1}$, $\bar{\mathbf{x}} = \mathbf{Lx}$ and $\bar{\mathbf{y}} = \mathbf{y}$ and solve (5.11) for $\bar{\mathbf{x}}$. The solution to the original problem (5.10) is found from the transformation $\mathbf{x} = \mathbf{L}^{-1}\bar{\mathbf{x}}$. Applying the inverse of $\mathbf{L}$ should be done either with a precalculated factorization or by an iterative method depending on the size and type of $\mathbf{L}$. If $\mathbf{L}$ has a non-trivial null-space $\mathcal{N}(\mathbf{L})$ we must take it into consideration as we demonstrate later by using the weighted pseudo-inverse.

Obviously components in the null-space of $\mathbf{L}$ are not regularized by the Tikhonov method and just using the standard Moore-Penrose pseudo-inverse to transform the problem is not an option as we will explain later. The answer is the "**K**-weighted pseudo-inverse of **L**" [35] defined by

$$\mathbf{L}_{\mathbf{K}}^{\dagger} = (\mathbf{I} - (\mathbf{K}(\mathbf{I} - \mathbf{L}^{\dagger}\mathbf{L}))^{\dagger}\mathbf{K})\mathbf{L}^{\dagger}. \tag{5.12}$$

We will need the solution in the null-space of $\mathbf{L}$ (the unregularized part of the solution) given by

$$\mathbf{x}_0 = (\mathbf{K}(\mathbf{I} - \mathbf{L}^{\dagger}\mathbf{L}))^{\dagger}\mathbf{y}, \tag{5.13}$$

where $(\mathbf{I} - \mathbf{L}^{\dagger}\mathbf{L})$ is a projection into the null-space of $\mathbf{L}$.

The transformation is then done by using

$$\bar{\mathbf{K}} = \mathbf{K}\mathbf{L}_{\mathbf{K}}^{\dagger}, \quad \bar{\mathbf{y}} = \mathbf{y} - \mathbf{K}\mathbf{x}_0, \tag{5.14}$$

The solution to the general form Tikhonov problem (5.10) is obtained from the solution of (5.11) via the transformation

$$\mathbf{x} = \mathbf{L}_{\mathbf{K}}^{\dagger}\bar{\mathbf{x}} + \mathbf{x}_0.$$

In (2.8) we have seen the usual Moore-Penrose pseudo-inverse expressed in terms of the SVD. Assuming that $\mathbf{K} \in \mathbb{R}^{m \times n}$, $m \geq n$ and that $\mathcal{N}(\mathbf{K}) \cap \mathcal{N}(\mathbf{L}) = \{0\}$ we have a similar expression for the weighted pseudo-inverse using the GSVD

$$\mathbf{L}_{\mathbf{K}}^{\dagger} = \mathbf{X}\mathbf{M}^{\dagger}\mathbf{V}^T. \tag{5.15}$$

Why the Weighted Pseudo-Inverse?

The iterative Krylov methods CGLS and LSQR can be used to solve a general-form Tikhonov problem in two ways:

- Apply the iterative least squares solver to

$$\mathbf{x}_\lambda = \underset{\mathbf{x}}{\operatorname{argmin}} \left\| \begin{bmatrix} \mathbf{K} \\ \lambda \mathbf{L} \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} \right\|_2 ,$$

  so that the Krylov methods find solutions in the Krylov space

$$\mathcal{K}_k \left(\mathbf{K}^T \mathbf{K} + \lambda^2 \mathbf{L}^T \mathbf{L} \, , \mathbf{K}^T \mathbf{y} \right) .$$

  Note that the Krylov subspace $\mathcal{K}_k \left(\mathbf{K}^T \mathbf{K} + \lambda_1^2 \mathbf{L}^T \mathbf{L} \, , \mathbf{K}^T \mathbf{y} \right)$ is different from the Krylov space $\mathcal{K}_k \left(\mathbf{K}^T \mathbf{K} + \lambda_2^2 \mathbf{L}^T \mathbf{L} \, , \mathbf{K}^T \mathbf{y} \right)$ if $\lambda_1 \neq \lambda_2$ and $\mathbf{L} \neq \mathbf{I}$.

- Solve the least squares problem

$$\mathbf{x}_\lambda = \mathbf{L}_{\mathbf{K}}^\dagger \underset{\bar{\mathbf{x}}}{\operatorname{argmin}} \left\| \begin{bmatrix} \mathbf{K} \mathbf{L}_{\mathbf{K}}^\dagger \\ \lambda \mathbf{I} \end{bmatrix} \bar{\mathbf{x}} - \begin{bmatrix} \bar{\mathbf{y}} \\ \mathbf{0} \end{bmatrix} \right\|_2 ,$$

  where the Krylov space becomes

$$\mathcal{K}_k \left((\mathbf{L}_{\mathbf{K}}^\dagger)^T \mathbf{K}^T \mathbf{K} \mathbf{L}_{\mathbf{K}}^\dagger \, , (\mathbf{L}_{\mathbf{K}}^\dagger)^T \mathbf{K}^T \bar{\mathbf{y}} \right) .$$

  Observe that the Krylov subspace is independent of λ, because $\lambda^2 \mathbf{I}$ only shifts the spectrum. This approach makes it possible to solve for several λs simultaneously or do the damping without including the identity operator, a feature implemented in the original LSQR code [106].

  The standard form gives the possibility to use the parameter choice methods discussed in the previous section.

But why not simply use the standard Moore-Penrose pseudo-inverse? In the following we derive the weighted pseudo-inverse and in the process we see where the ordinary pseudo-inverse fails. The derivation is inspired by the similar process in Eldén [33]. We first set up the Tikhonov problem in its least squares formulation:

$$\min \left\| \begin{bmatrix} \mathbf{K} \\ \lambda \mathbf{L} \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} \right\|_2 .$$

We will assume that $\mathbf{L} \in \mathbb{R}^{p \times n}$ with $p \geq n$ and that $\mathbf{L}$ has a non-trivial null-space. Then the SVD of $\mathbf{L}$ takes the form

$$\mathbf{L} = \begin{bmatrix} \mathbf{U}_1 & \mathbf{U}_0 \end{bmatrix} \begin{bmatrix} \mathbf{\Sigma} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{V}_1 & \mathbf{V}_0 \end{bmatrix}^T , \tag{5.16}$$

where the columns of $\mathbf{V}_0 \in \mathbb{R}^{k\times n}$ span the null-space of $\mathbf{L}$. Inserting the transformation $\mathbf{x} = \mathbf{V}_1\mathbf{s} + \mathbf{V}_0\mathbf{t}$ into the least squares problem yields

$$\min \left\| \begin{bmatrix} \mathbf{K}\mathbf{V}_1 & \mathbf{K}\mathbf{V}_0 \\ \lambda\mathbf{L}\mathbf{V}_1 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ \mathbf{t} \end{bmatrix} - \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} \right\|_2 .$$

The next step is to compute the QR factorization

$$\mathbf{K}\mathbf{V}_0 = \mathbf{Q}_1\mathbf{R} = [\mathbf{Q}_1\, \mathbf{Q}_0] \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}, \tag{5.17}$$

where $\mathbf{R} \in \mathbb{R}^{k\times k}$ has full rank because we assume that the null-spaces of $\mathbf{K}$ and $\mathbf{L}$ intersect trivially. Multiplying the least squares system with the orthogonal matrix $\mathrm{diag}(\mathbf{Q}, \mathbf{I})$ gives

$$\min \left\| \begin{bmatrix} \mathbf{Q}_1^T\mathbf{K}\mathbf{V}_1 & \mathbf{Q}_1^T\mathbf{K}\mathbf{V}_0 \\ \mathbf{Q}_0^T\mathbf{K}\mathbf{V}_1 & \mathbf{0} \\ \lambda\mathbf{L}\mathbf{V}_1 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ \mathbf{t} \end{bmatrix} - \begin{bmatrix} \mathbf{Q}_1^T\mathbf{y} \\ \mathbf{Q}_0^T\mathbf{y} \\ \mathbf{0} \end{bmatrix} \right\|_2 .$$

Because $\mathbf{Q}_1^T\mathbf{K}\mathbf{V}_0 = \mathbf{R}$ has full rank we can find $\mathbf{t}$, such that the residual vector of the first row is zero, from

$$\mathbf{t} = \mathbf{R}^{-1}(\mathbf{Q}_1^T\mathbf{y} - \mathbf{Q}_1^T\mathbf{K}\mathbf{V}_1\mathbf{s}).$$

Thereby we have reduced the least squares system to

$$\min \left\| \begin{bmatrix} \mathbf{Q}_0^T\mathbf{K}\mathbf{V}_1 \\ \lambda\mathbf{L}\mathbf{V}_1 \end{bmatrix} \mathbf{s} - \begin{bmatrix} \mathbf{Q}_0^T\mathbf{y} \\ \mathbf{0} \end{bmatrix} \right\|_2 .$$

Finally, we use the SVD of $\mathbf{L}$ (see (5.16)) and set $\mathbf{z} = \mathbf{L}\mathbf{V}_1\mathbf{s} = \mathbf{U}_1\mathbf{\Sigma}\mathbf{s}$ which we substitute into the least squares problem

$$\min \left\| \begin{bmatrix} \mathbf{Q}_0^T\mathbf{K}\mathbf{V}_1\mathbf{\Sigma}^{-1}\mathbf{U}_1^T \\ \lambda\mathbf{I} \end{bmatrix} \mathbf{z} - \begin{bmatrix} \mathbf{Q}_0^T\mathbf{y} \\ \mathbf{0} \end{bmatrix} \right\|_2 = \min \left\| \begin{bmatrix} \mathbf{Q}_0^T\mathbf{K}\mathbf{L}^\dagger \\ \lambda\mathbf{I} \end{bmatrix} \mathbf{z} - \begin{bmatrix} \mathbf{Q}_0^T\mathbf{y} \\ \mathbf{0} \end{bmatrix} \right\|_2 . \tag{5.18}$$

That is, the Moore-Penrose pseudo-inverse cannot be used as standard-form transformation because the range of $\mathbf{K}\mathbf{L}^\dagger$ is not orthogonal to the range of $\mathbf{K}\mathbf{V}_0$. In (5.18) we remove any components in the range of $\mathbf{K}\mathbf{V}_0$ (spanned by the columns of $\mathbf{Q}_1$) from the least squares problem through the multiplication of $\mathbf{Q}_0^T$. If we can modify the pseudo-inverse so that $\min\|\mathbf{K}\mathbf{P}\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0^T\mathbf{y}\|_2 = \min\|\mathbf{Q}_0^T\mathbf{K}\mathbf{L}^\dagger\mathbf{t} - \mathbf{Q}_0^T\mathbf{y}\|_2$ and $\mathbf{L}\mathbf{P}\mathbf{L}^\dagger = \mathbf{I}$ we have a useful transformation:

$$\begin{aligned} \min\|\mathbf{Q}_0^T\mathbf{K}\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0^T\mathbf{y}\|_2 &= \min\|\mathbf{Q}_0\mathbf{Q}_0^T\mathbf{K}\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0\mathbf{Q}_0^T\mathbf{y}\|_2 \\ &= \min\|(\mathbf{I} - \mathbf{Q}_1\mathbf{Q}_1^T)\mathbf{K}\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0\mathbf{Q}_0^T\mathbf{y}\|_2 \\ &= \min\|(\mathbf{K} - \mathbf{Q}_1\mathbf{Q}_1^T\mathbf{K})\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0\mathbf{Q}_0^T\mathbf{y}\|_2 \end{aligned}$$

and because $\mathbf{K}\mathbf{V}_0(\mathbf{K}\mathbf{V}_0)^\dagger = \mathbf{Q}_1\mathbf{Q}_1^T$ we get

$$\begin{aligned}
\min\|\mathbf{Q}_0^T\mathbf{K}\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0^T\mathbf{y}\|_2 &= \min\|(\mathbf{K} - \mathbf{K}\mathbf{V}_0(\mathbf{K}\mathbf{V}_0)^\dagger\mathbf{K})\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0\mathbf{Q}_0^T\mathbf{y}\|_2 \\
&= \min\|\mathbf{K}(\mathbf{I} - \mathbf{V}_0(\mathbf{K}\mathbf{V}_0)^\dagger\mathbf{K})\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0\mathbf{Q}_0^T\mathbf{y}\|_2 \\
&= \min\|\mathbf{K}(\mathbf{I} - (\mathbf{K}\mathbf{V}_0\mathbf{V}_0^T)^\dagger\mathbf{K})\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0\mathbf{Q}_0^T\mathbf{y}\|_2 \\
&= \min\|\mathbf{K}(\mathbf{I} - (\mathbf{K}(\mathbf{I} - \mathbf{L}^\dagger\mathbf{L}))^\dagger\mathbf{K})\mathbf{L}^\dagger\mathbf{z} - \mathbf{Q}_0\mathbf{Q}_0^T\mathbf{y}\|_2 \\
&= \min\|\mathbf{K}\mathbf{L}_\mathbf{K}^\dagger\mathbf{z} - \mathbf{Q}_0\mathbf{Q}_0^T\mathbf{y}\|_2,
\end{aligned}$$

where have the $\mathbf{K}$-weighted pseudo-inverse $\mathbf{L}_\mathbf{K}^\dagger = (\mathbf{I} - (\mathbf{K}(\mathbf{I} - \mathbf{L}^\dagger\mathbf{L})^\dagger\mathbf{K})\mathbf{L}^\dagger$. Also note that $\mathbf{L}(\mathbf{I} - (\mathbf{K}(\mathbf{I} - \mathbf{L}^\dagger\mathbf{L}))^\dagger\mathbf{K})\mathbf{L}^\dagger = \mathbf{V}_1\mathbf{V}_1^T + \mathbf{V}_0\mathbf{V}_0^T = \mathbf{I}$.

5.2.1 Evaluation of the Weighted Pseudo-Inverse

Most previous work on how to evaluate the pseudo-inverse involves assumptions that we in some cases are unable to fulfill partly due to the object oriented approach. In Eldén [33] two QR factorizations are used to form the explicit transformation $\bar{\mathbf{K}} = \mathbf{Q}_2^T\mathbf{K}\mathbf{L}^\dagger$ and $\bar{\mathbf{y}} = \mathbf{Q}_2^T\mathbf{y}$, as seen in (5.18). The implicit transformation used in the previous toolbox [68] assumes that the regularization matrix $\mathbf{L} \in \mathbb{R}^{p\times n}$ has $p \leq n$ and that we can partition $\mathbf{L}$ into two parts. However, with objects we cannot always perform QR factorizations and, as we will see later, we also wish to solve problems where $\mathbf{L}$ has (many) more rows than columns—but still with a non-trivial null-space.

The approach described next assumes nothing about the size of $\mathbf{L}$ and does not require a factorization of $\mathbf{L}$. However, it requires that the we know $\mathbf{V}_0$, that is, the null-space $\mathcal{N}(\mathbf{L})$, and that it is of relatively small dimension. In regularization problems it is usually of interest that the null-space of $\mathbf{L}$ is small. Otherwise we have large parts of the solution outside the reach of the regularization. Furthermore, we utilize the usual pseudo-inverse $\mathbf{L}^\dagger$, that is, we compute least squares solutions with $\mathbf{L}$. The least squares solution can be computed through direct or iterative methods depending on the particular choice of $\mathbf{L}$. Finally, we require that $\mathcal{N}(\mathbf{L})\cap\mathcal{N}(\mathbf{K}) = \{0\}$, so that the Tikhonov solution is unique.

We evaluate $\mathbf{L}_\mathbf{K}^\dagger$ as a mix of the explicit and implicit transform of [66, § 2.3.1–2.3.2]. If we have the null-space spanned by the orthogonal operator $\mathbf{V}_0$ the inner pseudo-inverse of (5.12) can be written

$$\begin{aligned}
(\mathbf{K}(\mathbf{I} - \mathbf{L}^\dagger\mathbf{L}))^\dagger &= (\mathbf{K}\mathbf{V}_0\mathbf{V}_0^T)^\dagger \\
&= \mathbf{V}_0(\mathbf{K}\mathbf{V}_0)^\dagger \\
&= \mathbf{V}_0\mathbf{R}^{-1}\mathbf{Q}^T,
\end{aligned}$$

where $\mathbf{Q}\mathbf{R} = \mathbf{K}\mathbf{V}_0$ is a thin QR factorization seen in (5.17). Owing to $\mathbf{V}_0$'s small number of columns it is feasible to compute the QR factorization of $\mathbf{K}\mathbf{V}_0$. The null-space component (5.13) is calculated by

$$\mathbf{x}_0 = \mathbf{V}_0\mathbf{R}^{-1}\mathbf{Q}^T\mathbf{y}.$$

The computation of the weighted pseudo-inverse is now

$$\mathbf{L}_{\mathbf{K}}^{\dagger} = (\mathbf{I} - \mathbf{V}_0\mathbf{R}^{-1}\mathbf{Q}^T)\mathbf{L}^{\dagger}, \tag{5.19}$$

that is, first we solve a least squares problem with $\mathbf{L}$ followed by a projection step. The transpose of (5.19) is

$$\left(\mathbf{L}_{\mathbf{K}}^{\dagger}\right)^T = \left(\mathbf{L}^{\dagger}\right)^T (\mathbf{I} - \mathbf{Q}\mathbf{R}^{-T}\mathbf{V}_0^T). \tag{5.20}$$

To illustrate the procedure we consider the regularization operator

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_2 \otimes \mathbf{I} \\ \mathbf{I} \otimes \mathbf{L}_2 \end{bmatrix}. \tag{5.21}$$

The null-space of $\mathbf{L}_2$ is spanned by $\mathbf{w}_1 = [1 \dots 1]^T$ and $\mathbf{w}_2 = [1\ 2\ \dots\ n]^T$. A small calculation shows that the null-space of $\mathbf{L}$ is spanned by the columns of the Kronecker product $\mathbf{V}_0 = ([\mathbf{w}_1\ \mathbf{w}_2] \otimes [\mathbf{w}_1\ \mathbf{w}_2]) \in \mathbb{R}^{n^2 \times 4}$. In this case a QR factorization of $\mathbf{K}\mathbf{V}_0$ would be a matter of performing 4 Householder reflections. However, if $\mathbf{K}$ is a Kronecker product (with terms of dimensions that allow for the multiplication with $\mathbf{V}_0$) the QR factorization can be done even smarter

$$\begin{aligned}(\mathbf{K}_1 \otimes \mathbf{K}_2)(\mathbf{W}_1 \otimes \mathbf{W}_1) &= (\mathbf{K}_1\mathbf{W}_1) \otimes (\mathbf{K}_2\mathbf{W}_2) \\ &= (\mathbf{Q}_1\mathbf{R}_1) \otimes (\mathbf{Q}_2\mathbf{R}_2) \\ &= (\mathbf{Q}_1 \otimes \mathbf{Q}_2)(\mathbf{R}_1 \otimes \mathbf{R}_2),\end{aligned}$$

requiring two QR factorizations of two small matrices.

Finally, operations with the pseudo-inverse $\mathbf{L}^{\dagger}$ can be done by solving a least squares system with LSQR or CGLS, maybe accelerated using a preconditioner provided by the user.

Details from the Implementation

The class WeightedPseudoInverse implements the following methods

`WeightedPseudoInverse`**.** The weighted pseudo-inverse is a linear operator and is therefore implemented as a child class of LinearOperator. The constructor takes the arguments $\mathbf{K}$, $\mathbf{L}$ and $\mathbf{V}_0$ and stores them in the fields of the object. It also computes and stores the thin QR factorization $\mathbf{Q}\mathbf{R} = \mathbf{K}\mathbf{V}_0$.

`sub_getmatrix`**.** Computes a matrix representation of the weighted pseudo-inverse based on `getmatrix` of $\mathbf{K}$, $\mathbf{L}^{\dagger}$, $\mathbf{V}_0$, $\mathbf{Q}$ and $\mathbf{R}$.

`sub_applytovector`**.** Computes the multiplication of the weighted pseudo-inverse following the recipes in (5.19) and (5.20). To illustrate the connection we list the relevant lines

```
if gettransposed(LKinv)
  v = v - LKinv.K'*(LKinv.Q*(LKinv.R'\(LKinv.V0'*v)));
  v = LKinv.L' \ v;
else
  v  = LKinv.L \ v;
  v  = v - LKinv.V0*(LKinv.R\(LKinv.Q'*(LKinv.K*v)));;
end
```

where `LKinv` denotes the object and `LKinv.K` denotes the field holding $\mathbf{K}$.

Note that the (ordinary Moore-Penrose) pseudo-inverse of $\mathbf{L}$ is computed via the "backslash" operator. All classes are required to return the least squares solution of minimum norm for its backslash operator. Matlab calls `mldivide` which use `sub_solve` of the particular $\mathbf{L}$ object to compute the pseudo-inverse.

`nullcomp`. Return the solution component in the null-space of $\mathbf{L}$ according to (5.13).

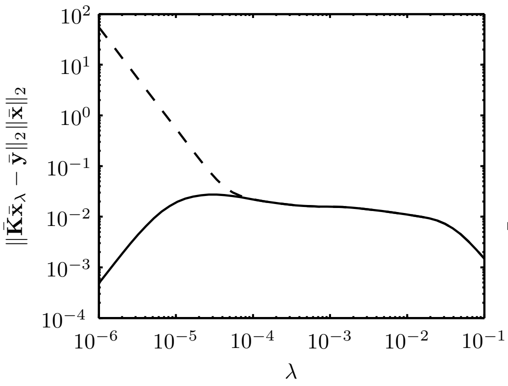
The evaluation of the $\mathbf{K}$-weighted pseudo-inverse depends on the possibility of computing the QR factorization of $\mathbf{K}\mathbf{V}_0$. However, as we saw in § 4.6.1 we are not always able to compute it (with the M$\mathcal{OO}$Re Tools toolbox in its current form). However, in our particular example we are able to compute the QR factorization of $\mathbf{K}\mathbf{V}_0$ because we have a Kronecker product and not, for example, a stacked operator.

Standard Form and Bounding Methods

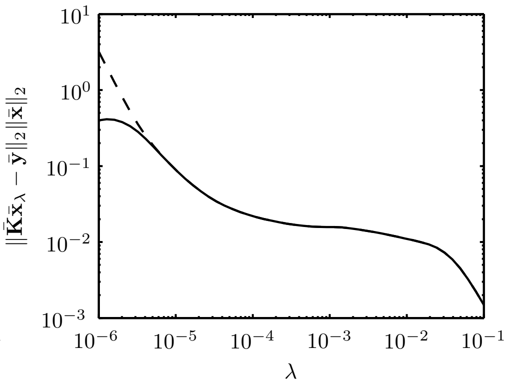
The standard-form transformation paves the way for the bounding methods that rely on a standard-form Tikhonov problem. In Fig. 5.5 we combine the standard-form transformation with Regińska's L-curve method bounds. We use the regularization matrix (5.21) and a test-problem created from a Kronecker product of two DERIV2 matrices. The commands used to create the weighted pseudo-inverse and the transformed system for this plot are found in the tutorial, see App. A.3. From Fig. 5.5(a) we see a result where the bounds are far apart. Increasing the tolerance gives better bounds as seen in Fig. 5.5(b). However, the result does not reveal a minimum. If we step back and plot the regular L-curve, see Fig. 5.5(c) we do indeed see the L-curve with a corner. Changing the "rotation" slightly and displaying $\|\bar{\mathbf{K}}\bar{\mathbf{x}}_\lambda - \bar{\mathbf{y}}\|_2^2\|\bar{\mathbf{x}}\|_2$ yields the plot in Fig. 5.5(d) where we see a nice minimum that corresponds to the corner seen on the L-curve.

5.3 A Tikhonov Preconditioner

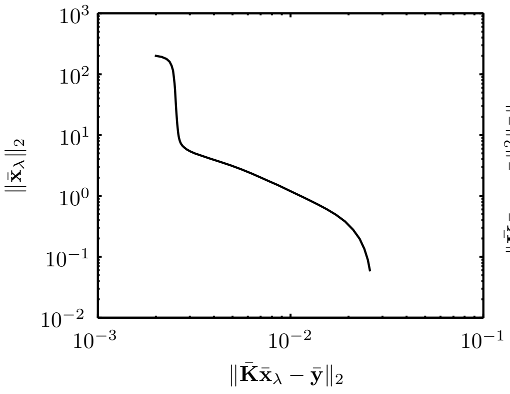
The objective of preconditioning ill-posed problems can be quite different from the usual preconditioning setting. Applying a preconditioner to a Krylov subspace method and relying on semi-convergence, that is, regularization by projection, can be dangerous. The preconditioner might introduce components where the noise dominates before the undisturbed components. That is, the preconditioner should improve conditioning of the operator for the components that are undisturbed by noise. The preconditioner by Hanke, Nagy and Plemmons [61] and the related preconditioner



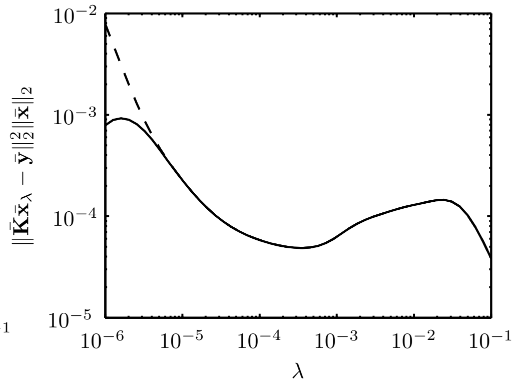
(a) $\tau = 2$ and 19 iterations



(b) $\tau = 0.1$ and 55 iterations



(c) $\tau = 0.1$ and 55 iterations



(d) $\tau = 0.1$ and 55 iterations

Figure 5.5: Standard-form transformation and Regińska's L-curve. The problem is constructed from a Kronecker product $\mathbf{K} = \mathbf{K}_1 \otimes \mathbf{K}_2 \in \mathbb{R}^{1600\times 1600}$ of two DERIV2 problems $\mathbf{K}_1, \mathbf{K}_2 \in \mathbb{R}^{40\times 40}$. The regularization matrix is the block matrix (5.21) and we have $\mathbf{L} \in \mathbb{R}^{3120\times 1600}$. The solution $\mathbf{x}$ is the usual BLUR solution and the right-hand side is constructed from $\mathbf{y} = \mathbf{K}\mathbf{x} + \mathbf{e}$, where $\mathbf{e}$ contains normal distributed noise with norm $10^{-2}\|\mathbf{y}\|_2$. Each iteration of the bidiagonalization involves two applications of the weighted pseudo-inverse, where we need to solve least squares problems of the type $\min\|\mathbf{L}\mathbf{x} - \mathbf{y}\|_2$ and $\min\|\mathbf{L}^T\mathbf{x} - \mathbf{y}\|_2$ via LSQR. We used the stopping criterion 5.9.

by Kilmer [86] are examples of preconditioners that, properly constructed, improve the conditioning of the space containing the undisturbed components, that is, components corresponding to large singular values. Both preconditioners are aimed at the class of block Toeplitz with Toeplitz block matrices, see also [24].

Preconditioning of a Tikhonov regularized problem is, on the other hand, straightforward as we apply the traditional preconditioning idea of approximating the perfect preconditioner $\mathbf{P}^T\mathbf{P}$,

$$\mathbf{P}^T\mathbf{P} = (\mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{L}^T\mathbf{L}),$$

where the $\mathbf{P}$ can be used in e.g. LSQR and $\mathbf{P}^T\mathbf{P}$ in ordinary CG. Such preconditioners are seen in [22, 23] where circulant approximations to $[\mathbf{K}^T\ \lambda\mathbf{L}^T]^T$ are constructed. In [84] $\mathbf{K}$ is approximated by a Kronecker product and applied to a standard-form Tikhonov problem. The preconditioner in the following is similar to the just mentioned ideas.

In some cases we can calculate a regularized solution from a standard-form Tikhonov problem very easily. For instance, if we have a Kronecker product $\mathbf{K} = \mathbf{K}_1 \otimes \mathbf{K}_2$ we can easily compute the SVD from two SVDs of small matrices. Utilizing the SVD we can calculate solutions to the standard-form Tikhonov problems and the idea is to apply the solver for the standard problem as a preconditioner for the general-form Tikhonov problem, where the GSVD might be unavailable.

To discuss the preconditioner we will need the normal equation system for the general-form Tikhonov problem

$$\mathbf{A}_\lambda = \mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{L}^T\mathbf{L} = \widehat{\mathbf{X}}^{-T}(\widehat{\mathbf{\Sigma}}^2 + \lambda^2\widehat{\mathbf{M}}^2)\widehat{\mathbf{X}}^{-1}, \tag{5.22}$$

where the system is also written in terms of the GSVD of $(\mathbf{K}, \mathbf{L})$, cf. Def. 2.3. The preconditioner is based on solving the standard-form problem via the SVD;

$$\begin{aligned} \mathbf{P}_\gamma^T\mathbf{P}_\gamma &= (\mathbf{K}^T\mathbf{K} + \gamma^2\mathbf{I}) = \mathbf{V}(\mathbf{\Sigma}^2 + \gamma^2\mathbf{I})\mathbf{V}^T \\ \mathbf{P}_\gamma &= \mathbf{V}(\mathbf{\Sigma}^2 + \gamma^2\mathbf{I})^{1/2}\mathbf{V}^T \end{aligned} \tag{5.23}$$

where we expand the matrix in terms of the SVD of $\mathbf{K} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ and γ is a regularization parameter for the preconditioner.

Preconditioning of the Standard-Form Problem

We first investigate the trivial case of applying the preconditioner to the problem in standard form, that is, we have $\mathbf{L} = \mathbf{I}$, $\widehat{\mathbf{M}} = \mathbf{I}$, $\widehat{\mathbf{X}} = \mathbf{V}$ and $\widehat{\mathbf{\Sigma}} = \mathbf{\Sigma}$ in (5.22). We will bound the condition number, as the condition number can be used to estimate worst case convergence properties, see e.g. [46]. A condition number near 1 is desirable and a large condition number is a sign of trouble.

We can now calculate norms of the preconditioned system $\mathbf{S} = \mathbf{A}_\lambda \mathbf{P}_\gamma^{-2}$

$$\begin{aligned}
\|\mathbf{S}\|_2 &= \|\mathbf{A}_\lambda \mathbf{P}_\gamma^{-2}\|_2 \\
&= \|\mathbf{V}(\mathbf{\Sigma}^2 + \lambda^2\mathbf{I})\mathbf{V}^T\mathbf{V}(\mathbf{\Sigma}^2 + \gamma^2\mathbf{I})^{-1}\mathbf{V}^T\|_2 \\
&= \|(\mathbf{\Sigma}^2 + \lambda^2\mathbf{I})(\mathbf{\Sigma} + \gamma^2\mathbf{I})^{-1}\|_2 \\
&= \max_i \frac{\sigma_i^2 + \lambda^2}{\sigma_i^2 + \gamma^2} \\
&= \max_i \tilde{\sigma}_i^2,
\end{aligned} \tag{5.24}$$

where $\tilde{\sigma}_i^2 = (\sigma_i^2 + \lambda^2)/(\sigma_i^2 + \gamma^2)$ for $i = 1 \dots n$. Similarly we get for the inverse of the preconditioned system

$$\begin{aligned}
\|\mathbf{S}^{-1}\|_2 &= \|(\mathbf{\Sigma}^2 + \lambda^2\mathbf{I})^{-1}(\mathbf{\Sigma} + \gamma^2\mathbf{I})\|_2 \\
&= \max_i \frac{1}{\tilde{\sigma}_i^2}.
\end{aligned} \tag{5.25}$$

Assume that $\sigma_1 \gg \lambda, \gamma \gg \sigma_n$, then $\tilde{\sigma}_1^2 \approx 1$ and $\tilde{\sigma}_n^2 \approx \lambda^2/\gamma^2$. We now consider three possibilities:

- If $\lambda = \gamma$ we have $\tilde{\sigma}_i = 1$ and, as expected for the perfect preconditioner, $\mathrm{cond}(\mathbf{S}) = 1$.
- If $\lambda < \gamma$ the we have $\tilde{\sigma}_1 \geq \tilde{\sigma}_n$ giving the condition number
$$\begin{aligned}
\mathrm{cond}(\mathbf{S}) &= \|\mathbf{S}\|_2 \|\mathbf{S}^{-1}\|_2 = \tilde{\sigma}_1^2(1/\tilde{\sigma}_n^2) \\
&\approx \gamma^2/\lambda^2
\end{aligned}$$
- If $\lambda > \gamma$ we have that $\tilde{\sigma}_1 \leq \tilde{\sigma}_n$ giving the condition number
$$\begin{aligned}
\mathrm{cond}(\mathbf{S}) &= (1/\tilde{\sigma}_1)\tilde{\sigma}_n \\
&\approx \lambda^2/\gamma^2.
\end{aligned}$$

Thus, if γ is chosen close to λ we will have a good preconditioner. This is a quite unsurprising result as setting $\gamma = \lambda$ gives the perfect preconditioner.

5.3.1 The General Case

Motivated by the simple case we now look at the preconditioner applied to the general Tikhonov problem. We will pursue the same approach as in the previous section. Unfortunately the result is very discouraging with respect to getting small condition numbers. However, the line of thought is included as one of the "rarely reported" negative results, cf. the citation of the Stephen Wright in the beginning of the present chapter.

Let the diagonal of $\mathbf{\Phi} = (\widehat{\mathbf{\Sigma}}^2 + \lambda^2 \widehat{\mathbf{M}}^2) \in \mathbb{R}^{n\times n}$ and $\mathbf{\Psi} = (\mathbf{\Sigma}^2 + \gamma^2 \mathbf{I})^{-1} \in \mathbb{R}^{n\times n}$ be denoted $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$. The cosine of the angle between $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ is

$$\cos(\boldsymbol{\phi}, \boldsymbol{\psi}) = \frac{\boldsymbol{\phi}^T \boldsymbol{\psi}}{\|\boldsymbol{\phi}\|_2 \|\boldsymbol{\psi}\|_2}.$$

Since all elements of $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ are strictly positive they cannot be orthogonal and $\boldsymbol{\phi}^T \boldsymbol{\psi} > 0$ and therefore $\cos(\boldsymbol{\phi}, \boldsymbol{\psi}) > 0$. Hence, we can rearrange

$$\|\boldsymbol{\phi}\|_2 \|\boldsymbol{\psi}\|_2 = \frac{\boldsymbol{\phi}^T \boldsymbol{\psi}}{\cos(\boldsymbol{\phi}, \boldsymbol{\psi})}.$$

Because

$$\boldsymbol{\phi}^T \boldsymbol{\psi} = \sum_{i=1}^{n} \phi_i \psi_i \le n \max_i (\phi_i \psi_i) = n \|\mathbf{\Phi}\mathbf{\Psi}\|_2$$

we get the bound

$$\|\boldsymbol{\phi}\|_2 \|\boldsymbol{\psi}\|_2 \le \frac{n \|\mathbf{\Phi}\mathbf{\Psi}\|_2}{\cos(\boldsymbol{\phi}, \boldsymbol{\psi})}. \tag{5.26}$$

Now we look at the expression $\|\mathbf{\Phi}\mathbf{B}\mathbf{\Psi}\|_2 = \|\mathbf{B} \odot (\boldsymbol{\phi}\boldsymbol{\psi}^T)\|_2$, where $\odot$ is the Hadamard product (or elementwise multiplication). With Theorem 5.5.15 of Horn and Johnson [77] and the relation (5.26) we get

$$\begin{aligned} \|\mathbf{\Phi}\mathbf{B}\mathbf{\Psi}\|_2 &= \|\mathbf{B} \odot \boldsymbol{\phi}\boldsymbol{\psi}^T\|_2 \\ &\le \|\mathbf{B}\|_2 \|\boldsymbol{\phi}\boldsymbol{\psi}^T\|_2 \\ &= \|\mathbf{B}\|_2 \|\boldsymbol{\phi}\|_2 \|\boldsymbol{\psi}\|_2 \\ &\le \|\mathbf{B}\|_2 \frac{n \|\mathbf{\Phi}\mathbf{\Psi}\|_2}{\cos(\boldsymbol{\phi}, \boldsymbol{\psi})}. \end{aligned} \tag{5.27}$$

For $\|\mathbf{S}\|_2$ we have

$$\begin{aligned} \|\mathbf{S}\|_2 &= \|\widehat{\mathbf{X}}^{-T} (\widehat{\mathbf{\Sigma}}^2 + \lambda^2 \widehat{\mathbf{M}}^2) \widehat{\mathbf{X}}^{-1} \mathbf{V} (\mathbf{\Sigma}^2 + \gamma^2 \mathbf{I})^{-1} \mathbf{V}^T\|_2 \\ &\le \|\widehat{\mathbf{X}}^{-T}\|_2 \|(\widehat{\mathbf{\Sigma}}^2 + \lambda^2 \widehat{\mathbf{M}}^2) \widehat{\mathbf{X}}^{-1} \mathbf{V} (\mathbf{\Sigma}^2 + \gamma^2 \mathbf{I})^{-1}\|_2 \end{aligned}$$

and using our bound (5.27) to “unite” the two diagonal matrices and move $\widehat{\mathbf{X}}^{-1}\mathbf{V}$ out of the way we get

$$\begin{aligned} \|\mathbf{S}\|_2 &\le \frac{n}{\cos(\boldsymbol{\phi}, \boldsymbol{\psi})} \|\widehat{\mathbf{X}}^{-1}\|_2 \|\widehat{\mathbf{X}}^{-1}\mathbf{V}\|_2 \|(\widehat{\mathbf{\Sigma}}^2 + \lambda^2 \widehat{\mathbf{M}}^2)(\mathbf{\Sigma}^2 + \gamma^2 \mathbf{I})^{-1}\|_2 \\ &= \frac{n}{\cos(\boldsymbol{\phi}, \boldsymbol{\psi})} \|\widehat{\mathbf{X}}^{-1}\|_2^2 \|(\widehat{\mathbf{\Sigma}}^2 + \lambda^2 \widehat{\mathbf{M}}^2)(\mathbf{\Sigma}^2 + \gamma^2 \mathbf{I})^{-1}\|_2. \end{aligned}$$

Likewise we have for the norm of the inverse of $\mathbf{S}$

$$\|\mathbf{S}^{-1}\|_2 \leq \frac{n}{\cos(\boldsymbol{\phi},\boldsymbol{\psi})}\|\widehat{\mathbf{X}}\|_2^2\|(\widehat{\mathbf{\Sigma}}^2+\lambda^2\widehat{\mathbf{M}}^2)^{-1}(\mathbf{\Sigma}^2+\gamma^2\mathbf{I})\|_2.$$

We have now derived the following bound for the condition number of $\mathbf{S}$

$$\mathrm{cond}(\mathbf{S}) = \|\widehat{\mathbf{S}}\|_2\|\widehat{\mathbf{S}}^{-1}\|_2 \leq \left(\frac{n}{\cos(\boldsymbol{\phi},\boldsymbol{\psi})}\right)^2 \mathrm{cond}^2(\widehat{\mathbf{X}})\mathrm{cond}(\mathbf{\Omega}), \tag{5.28}$$

where $\mathbf{\Omega}$ is the diagonal matrix

$$\begin{aligned}\mathbf{\Omega} &= (\widehat{\mathbf{\Sigma}}^2+\lambda^2\widehat{\mathbf{M}}^2)(\mathbf{\Sigma}^2+\gamma^2\mathbf{I})^{-1}\\ &= (\widehat{\mathbf{\Sigma}}^2(1-\lambda^2)+\lambda^2\mathbf{I})(\mathbf{\Sigma}^2+\gamma^2\mathbf{I})^{-1}.\end{aligned}$$

The diagonal entries of $\mathbf{\Omega}$ are

$$\omega_i = \frac{\widehat{\sigma}_i^2(1-\lambda^2)+\lambda^2}{\sigma_i^2+\gamma^2} = \frac{\widehat{\sigma}_i^2 p+\lambda^2}{\sigma_i^2+\gamma^2}, \tag{5.29}$$

where $p = (1-\lambda^2)$ is introduced to shorten the following expressions.

From [63, Theorem 2.4] we have the bounds on how the singular values change depending on the particular $\widehat{\mathbf{X}}$ (and thereby also the regularization matrix $\mathbf{L}$):

$$1/\|\widehat{\mathbf{X}}\|_2 \leq \frac{\widehat{\sigma}_i}{\sigma_i} \leq \|\widehat{\mathbf{X}}^{-1}\|_2 \;\Leftrightarrow\; \sigma_i/\|\widehat{\mathbf{X}}\|_2 \leq \widehat{\sigma}_i \leq \sigma_i\|\widehat{\mathbf{X}}^{-1}\|_2. \tag{5.30}$$

Inserted into (5.29) we get the following bounds on the extreme values of ω_i

$$\begin{aligned}\omega_1^- &= \frac{p}{\|\widehat{\mathbf{X}}\|_2^2}\frac{\sigma_1^2+\lambda^2\|\widehat{\mathbf{X}}\|_2^2/p}{\sigma_1^2+\gamma^2} \leq \omega_1 \leq \|\widehat{\mathbf{X}}^{-1}\|_2^2 p\frac{\sigma_1^2+\lambda^2/(\|\widehat{\mathbf{X}}^{-1}\|_2^2 p)}{\sigma_1^2+\gamma^2} = \omega_1^+\\ \omega_n^- &= \frac{p}{\|\widehat{\mathbf{X}}\|_2^2}\frac{\sigma_n^2+\lambda^2\|\widehat{\mathbf{X}}\|_2^2/p}{\sigma_n^2+\gamma^2} \leq \omega_n \leq \|\widehat{\mathbf{X}}^{-1}\|_2^2 p\frac{\sigma_n^2+\lambda^2/(\|\widehat{\mathbf{X}}^{-1}\|_2^2 p)}{\sigma_n^2+\gamma^2} = \omega_n^+\end{aligned}$$

These bounds on ω_1 and ω_n tell us that

$$\|\mathbf{\Omega}\|_2 \leq \max\left\{\omega_1^+,\omega_n^+\right\} \tag{5.31}$$

$$\|\mathbf{\Omega}^{-1}\|_2 \leq \max\left\{1/\omega_n^-,1/\omega_1^-\right\} \tag{5.32}$$

and therefore

$$\mathrm{cond}(\mathbf{\Omega}) \leq \max\left\{\omega_1^+,\omega_n^+\right\}\max\left\{1/\omega_n^-,1/\omega_1^-\right\} \tag{5.33}$$

If we assume that $\lambda \ll 1$ then $p \approx 1$. If furthermore $\sigma_1 \gg \lambda^2\|\widehat{\mathbf{X}}\|_2^2, \lambda^2/\|\widehat{\mathbf{X}}^{-1}\|_2^2 \gg \sigma_n$ we have the following approximations

$$\begin{aligned}
\omega_1^- &\approx \frac{1}{\|\widehat{\mathbf{X}}\|_2^2}\frac{\sigma_1^2+\lambda^2\|\widehat{\mathbf{X}}\|_2^2}{\sigma_1^2+\gamma^2} \approx \frac{1}{\|\widehat{\mathbf{X}}\|_2^2}\frac{\sigma_1}{\sigma_1} = \frac{1}{\|\widehat{\mathbf{X}}\|_2^2},\\
\omega_1^+ &\approx \|\widehat{\mathbf{X}}^{-1}\|_2^2\frac{\sigma_1^2+\lambda^2/\|\widehat{\mathbf{X}}^{-1}\|_2^2}{\sigma_1^2+\gamma^2} \approx \|\widehat{\mathbf{X}}^{-1}\|_2^2\frac{\sigma_1^2}{\sigma_1^2} = \|\widehat{\mathbf{X}}^{-1}\|_2^2\\
\omega_n^- &\approx \frac{1}{\|\widehat{\mathbf{X}}\|_2^2}\frac{\sigma_n^2+\lambda^2\|\widehat{\mathbf{X}}\|_2^2}{\sigma_n^2+\gamma^2} \approx \frac{1}{\|\widehat{\mathbf{X}}\|_2^2}\frac{\lambda^2/\|\widehat{\mathbf{X}}\|_2^2}{\gamma^2} = \frac{\lambda^2}{\gamma^2}\\
\omega_n^+ &\approx \|\widehat{\mathbf{X}}^{-1}\|_2^2\frac{\sigma_n^2+\lambda^2/\|\widehat{\mathbf{X}}^{-1}\|_2^2}{\sigma_n^2+\gamma^2} \approx \|\widehat{\mathbf{X}}^{-1}\|_2^2\frac{\lambda^2/\|\widehat{\mathbf{X}}^{-1}\|_2^2}{\gamma^2} = \frac{\lambda^2}{\gamma^2}.
\end{aligned}$$

and by insertion we get an approximation to an upper bound on the condition number of $\mathbf{\Omega}$

$$\mathrm{cond}(\mathbf{\Omega}) \lessapprox \max\left\{\|\widehat{\mathbf{X}}^{-1}\|_2^2, \lambda^2/\gamma^2\right\} \max\left\{\gamma^2/\lambda^2, \|\widehat{\mathbf{X}}\|_2^2\right\}.$$

Assuming $\|\widehat{\mathbf{X}}\|_2 > \gamma^2/\lambda^2$ and $\|\widehat{\mathbf{X}}^{-1}\|_2 > \lambda^2/\gamma^2$ (likely if we select $\lambda = \gamma$ and $\mathbf{L}$ is properly scaled, see [63]) we get the desired bound on the condition number of the preconditioned system

$$\mathrm{cond}(\mathbf{S}) \lessapprox \left(\frac{n}{\cos(\boldsymbol{\phi}, \boldsymbol{\psi})}\right)^2 \mathrm{cond}^4(\widehat{\mathbf{X}}).$$

This bound is not very satisfactory as we have not found a good way of bounding $\cos(\boldsymbol{\phi}, \boldsymbol{\psi})$. Considering the expected convergence speed the bound is indeed very discouraging with the squared term, $n^2/\cos^2(\boldsymbol{\phi}, \boldsymbol{\psi})$, appearing. However, experiments similar to those in the following have all shown the bounds to be very pessimistic for all 1-D test problems. Furthermore, keep in mind that the condition number is only part of a worst case convergence estimate. Not only the ratio of largest and smallest singular value is interesting but also any clustering of the singular values and the components in the particular right-hand side.

5.3.2 Numerical Experiments

We have conducted a number of experiments with the preconditioner to illustrate the theory of the previous section. First we take a look at the condition numbers of small systems. Then we look at three examples of eigenvalue distributions for three different choices of the regularization parameter γ in the preconditioner. Finally we look at the actual convergence for a large-scale BLUR problem.

Condition Numbers

In Fig. 5.6 we have used a small test problem, so that the condition number of a preconditioned system is easily computed. We use the preconditioner on a standard-form problem with the identity as regularization matrix and on problems with approximations to the first derivative, the second derivative and the first and second derivative stacked as the regularization matrix. The first case illustrates the warm up section while the second case illustrates an actual use of the preconditioner. In Fig. 5.6(a) we use the same regularization matrix for problem and preconditioner and naturally we get perfect preconditioning, i.e., $\mathrm{cond}(\mathbf{A}_\lambda \mathbf{P}_\gamma^{-2}) = 1$ when the regularization parameters λ and γ are equal. Applying the preconditioner to a general-form Tikhonov problem shows that we indeed are able to improve the conditioning of the problem. We also see that the bounds derived in the previous section are extremely pessimistic as the $n^2\mathrm{cond}^4(\widehat{\mathbf{X}})$ (even omitting the cosine term) is above both the unpreconditioned system and the chosen limits of the plot for all three choices of regularization matrix $\mathbf{L}$.

Eigenvalues of Preconditioned System

The condition number of the preconditioned system is used for a worst case convergence bound for the conjugate gradient based algorithms. Any clustering of eigenvalues increases the convergence speed.

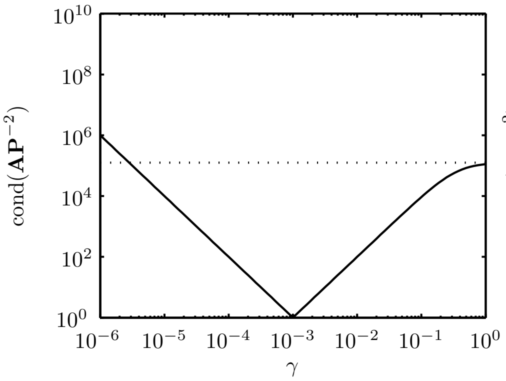
Intuitively the large singular values away from λ^2 are “unharmed” by the regularization term. Our preconditioner is also “unharmed” in this area of the spectrum and we could expect the resulting system to have a cluster of singular values near one for these components.

In the following we return to the regularization operator

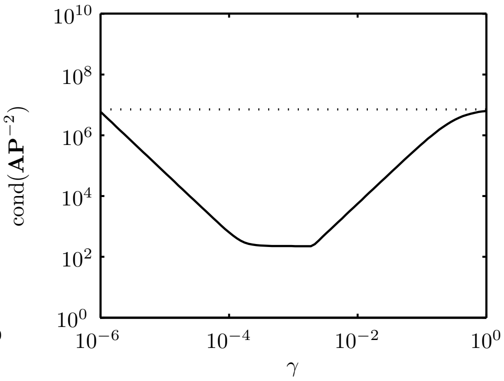
$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_1 \otimes \mathbf{I} \\ \mathbf{I} \otimes \mathbf{L}_1 \end{bmatrix}. \tag{5.34}$$

Even if $\mathbf{K}$ is a Kronecker prouduct, that is, if $\mathbf{K} = \mathbf{K}_1 \otimes \mathbf{K}_2$, we are not able to exploit the special structures to, for example, compute an GSVD. Therefore, if the dimension of the problem gets large we are not able to solve with a direct method and we must turn to an iterative method such as LSQR with preconditioning.

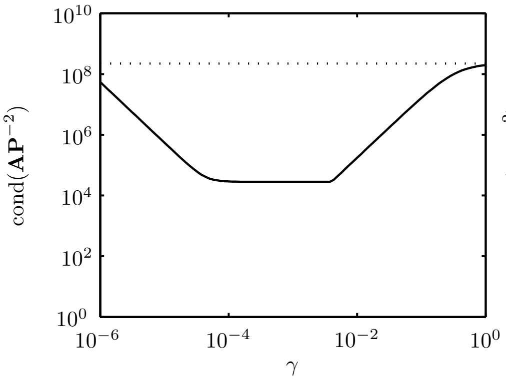
Figure 5.7 shows three examples of the eigenvalues of a general-form Tikhonov system and the corresponding preconditioned system. We see a cluster of unit eigenvalues in addition to the better conditioning of the system and it seems reasonable to expect faster convergence. The eigenvalues are computed for two choices of γ. The choice $\gamma^2 = \lambda^2$ makes the cluster of near unit eigenvalues appear as the largest eigenvalues whereas the choice $\gamma^2 = 10\lambda^2$ makes the eigenvalues appear as the smallest eigenvalues. As the Krylov subspace methods pick up components belonging to the largest/extreme eigenvalues first we will in the $\gamma^2 = 10\lambda^2$ case pick up the components belonging to the noise-space. This is not a problem as they are damped from the Tikhonov regularization assuming λ is chosen properly. However, the opposite behavior picking up the cluster first might be more desirable if λ is chosen too small and we need extra regularization from the projection. When $\gamma^2 \approx \|\mathbf{L}\|_2^2\lambda^2$ the situation is a mix of the above and the cluster lies in the middle of the spectrum, see Fig. 5.7(c).



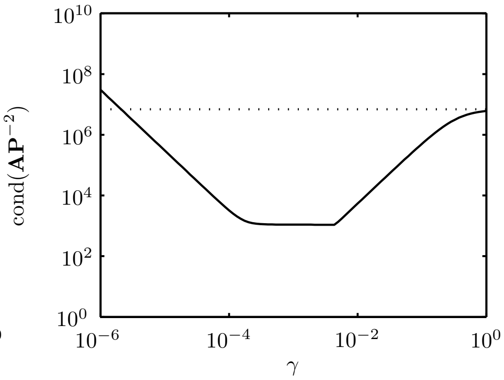
(a) $\mathbf{A} = \mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{I}$



(b) $\mathbf{A} = \mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{L}_1^T\mathbf{L}_1$

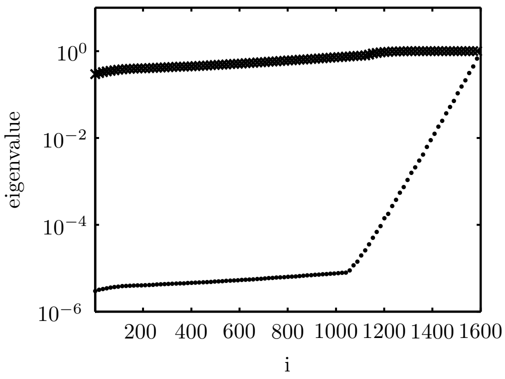


(c) $\mathbf{A} = \mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{L}_2^T\mathbf{L}_2$

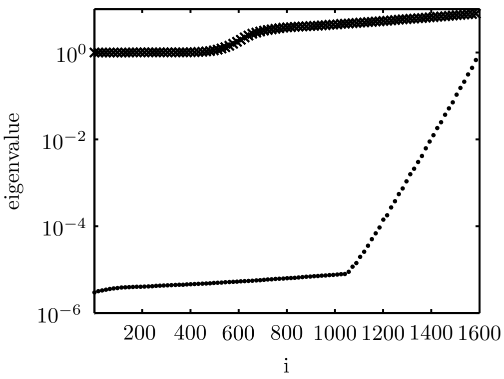


(d) $\mathbf{A} = \mathbf{K}^T\mathbf{K} + \lambda^2\mathbf{L}^T\mathbf{L}$

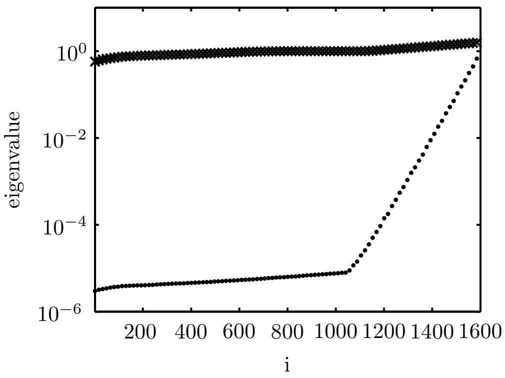
Figure 5.6: Condition numbers using different values of μ in preconditioner $\mathbf{P}_\gamma = (\mathbf{K}^T\mathbf{K} + \gamma^2\mathbf{I})^{-1/2}$. The test problem is HEAT with $n = 128$ with the identity $\mathbf{I}$ (a), the first derivative $\mathbf{L}_1$ (b), the second derivative $\mathbf{L}_2$ (c) and the stacked first and second derivative $\mathbf{L} = [\mathbf{L}_1^T\,\mathbf{L}_2]^T$ (d) as the regularization matrix. The regularization parameter $\lambda = 10^{-3}$ in both examples. The dotted line shows the condition number of the un-preconditioned system $\mathbf{A}$.



(a) $\gamma^2 = \lambda^2$



(b) $\gamma^2 = 10\lambda^2$



(c) $\gamma^2 = 5\lambda^2$

Figure 5.7: Eigenvalues of general form Tikhonov system (dots) and preconditioned Tikhonov systems (x) with different values of regularization parameter in preconditioner. The test problem is BLUR with $\sigma_x = 2$, $\sigma_y = 1.5$ (see Table D.1) and $n = 40$ giving an operator $\mathbf{K} \in \mathbb{R}^{1600\times1600}$. The regularization operator is the operator (5.34) and the regularization parameter is $\lambda = 10^{-3}$. Only every 16th eigenvalue is plotted.

Convergence Speed

The BLUR test problem tends to converge slowly for small regularization parameters λ. We continue with the regularization matrix (5.34) but increase the problem size to $n = 512^2$. This gives $\mathbf{K} = \mathbf{K}_1 \otimes \mathbf{K}_2 \in \mathbb{R}^{262144 \times 262144}$ and more importantly $\mathbf{L} \in \mathbb{R}^{523264 \times 262144}$ which, although sparse, makes it impossible to compute the GSVD. The SVD of the Kronecker product $\mathbf{K}$ poses no problem and the Tikhonov preconditioner is easily constructed.

Figure 5.8 shows the number of iterations needed to reduce the norm of the normal equation residual for a general-form Tikhonov problem of the form

$$\min \left\| \begin{bmatrix} \mathbf{K} \\ 10^{-3} \begin{bmatrix} (\mathbf{L}_1 \otimes \mathbf{I}) \\ (\mathbf{I} \otimes \mathbf{L}_1) \end{bmatrix} \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{y} \\ \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \end{bmatrix} \right\|_2 = \min \| \widehat{\mathbf{K}} \mathbf{x} - \widehat{\mathbf{y}} \|_2 .$$

That is, CGLS is stopped when $\|\widehat{\mathbf{K}}^T(\widehat{\mathbf{K}}\mathbf{x} - \widehat{\mathbf{y}})\|_2 < 10^{-10}\|\widehat{\mathbf{y}}\|_2$. It is seen that around $\gamma = 2 \cdot 10^{-3}$ we get the fastest convergence. In Fig. 5.9 we have plotted the norm of the normal equation residual for the general-form Tikhonov least squares problem for CGLS without preconditioning and with preconditioning using three different choices of γ. If no preconditioner is used the convergence is very slow. Extrapolation of the convergence seen from iteration 50 to 200 estimates that around 1300 iterations are needed. The preconditioner insures convergence to a normal equation residual precision of $\|\widehat{\mathbf{K}}^T(\widehat{\mathbf{K}}\mathbf{x}_k - \widehat{\mathbf{y}})\|_2 < 10^{-10}\|\widehat{\mathbf{y}}\|_2$ in just 7 iterations. However, the time to construct the preconditioner and the extra time used in each iteration to apply it must be considered also.

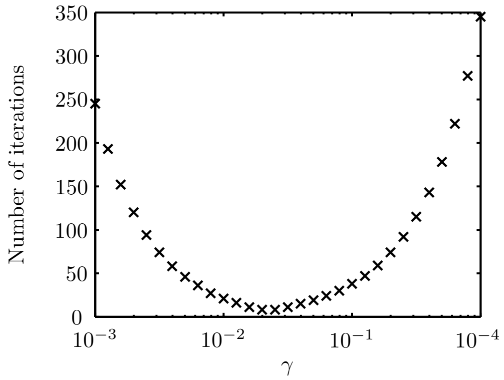


Figure 5.8: Number of iterations vs. the regularization parameter γ used in the preconditioner.

Table 5.1 shows timings collected during the experiment also seen in Fig. 5.9. Even though the construction phase is quite costly ($\approx$ 20 iterations of CGLS without preconditioning) we manage to get the solution very fast. Note that after 100 itera-

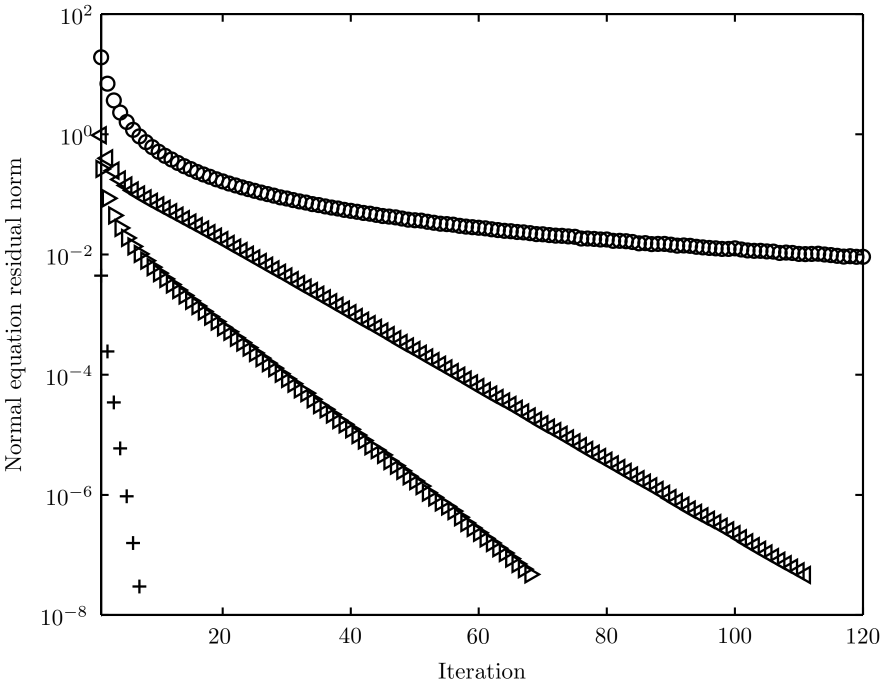


Figure 5.9: Convergence of CGLS without preconditioning (o) and with preconditioning with parameters $\gamma = 2 \cdot 10^{-2}$ (+), $\gamma = 2 \cdot 10^{-1}$ ($\triangleleft$) $\gamma = 2 \cdot 10^{-3}$ ($\triangleright$). The test problem is BLUR with $\sigma_x = \sigma_y = 1.5$ and $n = 512$ giving $\mathbf{K} \in \mathbb{R}^{512^2 \times 512^2}$. The regularization matrix is $\mathbf{L} = [(\mathbf{L}_1 \otimes \mathbf{I})^T \, (\mathbf{I} \otimes \mathbf{L}_1)^T]^T$ where $\mathbf{L}_1$ is an discrete approximation to the first derivative. The regularization parameter for the problem is $\lambda = 10^{-3}$. Timings for setup and iterations are listed in Table 5.1.

tions with the unpreconditioned system we are not nearly as close to the solution as the best of the preconditioned systems is after just one iteration.

	CGLS		Tikhonov preconditioned CGLS					
			$\gamma = 2 \cdot 10^{-1}$		$\gamma = 2 \cdot 10^{-2}$		$\gamma = 2 \cdot 10^{-3}$	
Phase	#	Time (s)	#	Time (s)	#	Time (s)	#	Time (s)
Setup		0		30		30		30
Iterations	$\gg$200	$\gg$333	68	265	7	29	111	438
Total		$\gg$333		295		59		468

Table 5.1: Timings with and without preconditioning for the convergence history plotted in Fig. 5.9. Note that CGLS without preconditioning does not converge to the same level of accuracy. Extrapolating from the convergence speed between iteration 50 and 200 estimates that round 1300 iterations, corresponding to 2100 seconds, are needed for CGLS without preconditioning to reach the same precision.

Implementation of the Preconditioner

The preconditioner is implemented in the TikhPrecond class that only implements a few methods:

`TikhPrecond`. The constructor takes the operator $\mathbf{K}$ and the regularization parameter γ as arguments. It computes the SVD $\mathbf{K} = \mathbf{U\Sigma V}^T$ and stores $\mathbf{V}$ and $(\mathbf{\Sigma}^T\mathbf{\Sigma} + \gamma^2\mathbf{I})^{-1/2}$ in fields of the object.

`sub_solve`. When the iterative algorithms solve with the preconditioner the method `LinearOperator/mldivide` calls the `sub_solve` method of the TikhPrecond class. The line computing $\mathbf{x} = \mathbf{P}_\gamma^{-1}\mathbf{b}$ of `sub_solve` is

```
x = P.V*(P.SS*(P.V'*b))
```

where `P` denotes the preconditioner object, `P.V` is the object containing the singular vectors and `P.SS` is the diagonal matrix computed by the constructor. Note that we do not consider whether the operator is transposed or not because the operator is symmetric (compare with the corresponding section on the weighted pseudo-inverse).

`sub_size`. Returns the size of the operator.

`sub_getmatrix`. Returns a double array representation of the operator.

5.4 Summary

In this chapter we have introduced and experimented with a stopping criterion for Lanczos bidiagonalization that forms the basis of the "bounding" parameter choice methods. The experiments were performed with the Regińska's variation of the L-curve method.

We have described how the standard-form transformation is implemented in the new toolbox. Because the bounding methods are based on the standard-form formulation it was natural to illustrate the standard-form transformation in connection with the bounding method of Regińska's L-curve.

Finally we have described a preconditioner for the general-form Tikhonov problem based on the standard-form problem. Despite discouraging theoretical bounds of worst case condition numbers and thereby worst case convergence we obtained good experimental results on the BLUR test problem.

CHAPTER 6

Conclusion

—filled with a liquid that was almost, but not quite, entirely unlike tea. Douglas Adams, "Hitchhikers Guide to the Galaxy" [1].

And this leads us to a suitable paradoxical conclusion. Even infinity—whichever one you choose—is not "the biggest thing that can exist". There's always something bigger still. But it's OK, you do learn to live with it eventually. Especially when you realise you can't live without it. Ian Stewart, "Never Ending Story" [120].

The goal of creating a modular and object oriented framework for solving discrete ill-posed problems has been achieved. The theory has been used to create the accompanying Matlab toolbox M$\mathcal{OO}$Re Tools.

The following is a summary of the results and observations made in this thesis.

6.1 Modular Algorithms

The regularization algorithms have been divided into 3 groups:

Penalty methods. The penalty methods regularize ill-posed problems by adding terms that discourage solutions with unwanted properties. The penalty functions are described as a sum of composite functions, where the first usually describes the model fit. Compared to the usual l_2-norm penalty functions, the use of other penalty functions on the model fit can make the method robust toward outliers in the data, while the use of other penalty functions with the regularization terms enables reconstruction of, for example, discontinuous solutions.

A flexible and modular algorithm is obtained through standardizing and extending the available penalty functions from just the l_2-norm penalty function.

We have through numerical experiments shown how regularized solutions with very different properties can be achieved with the modular approach.

Projection methods. The projection methods restrict a solution to lie in or to be orthogonal to a specified subspace. In this case we can also utilize the penalty functions to obtain solutions that are robust toward outliers or have discontinuities.

Essentially two variations of the projection methods are necessary. One module is required to minimize in a subspace, and one to minimize with a solution in a subspace orthogonal to another. Which one to choose depends on the particular subspace and formulation.

The basic iterative Krylov subspace methods MINRES, GMRES and LSQR can also be placed in this category, although they are restricted to minimizing in the l_2-norm.

Hybrid Methods. The Hybrid methods combine a projection method, often of the Krylov subspace type, with another regularization method on the smaller projected problem. The hybrid methods are the best showcase for modular methods as the idea inherently needs to combine at least two regularization methods, that can be conveniently provided as modules.

The choice of regularization parameter is largely dependent on the particular regularization method used. Hence, we have been unable to construct modules that can be used with all regularization routines. However, we can for a specific method, for example Tikhonov regularization, create modules that estimate the optimal regularization parameter.

6.2 Object Oriented Programming

We have analyzed the structures of the regularization problems and in particular the linear algebra "constructs" we need to set up and solve the problems.

Two basic concepts, the linear operator and the vector, form the basis of the object oriented framework. The interface of the linear operator defines what linear operators must be capable of, for example, a linear operator can be applied to a vector. This interface is defined in the class LinearOperator and similarly the class Vector defines the interface for all objects of the vector type. For some linear operators it is feasible to compute decompositions such as the SVD. In this case the linear operator also implements the interface defined by SVDOperator.

Using the interface we are able to exploit special structures and features of the operator without rewriting the actual regularization or minimization algorithm for each particular operator. We have demonstrated how the object oriented approach *automatically* enables CGLS to take advantage of the special properties of, for example, the Kronecker product where the fast multiplication can be exploited. Some regularization methods, such as Tikhonov regularization, exist in different versions;

one that operates with the interface of SVDOperator and one that operates with the LinearOperator interface. Thus, if the SVD is supported, then the SVD is used. Otherwise, the default LinearOperator version of Tikhonov regularization (based on damped LSQR) is used.

In addition to the problem-specific operators, we identified the most important generic operators faced in the regularization methods. For instance, we have classes representing the identity operator and block operators.

Inheritance makes it possible to extend or specialize objects. We have, for example, a general N-term Kronecker product class that we specialized to the often used two-term Kronecker product. In this process we can reuse code from the general case and redefine code to, for example, optimize multiplication for the simpler two-term case.

6.3 Other Results

In some cases we are not able to compute an SVD of the operator. Without the SVD it is time consuming to use the usual parameter choice methods for Tikhonov regularization as solutions must be computed for a range of regularization parameters. However, the "bounding" methods are able to compute upper and lower bounds on the necessary solution and residual norms, as well as bounds on the trace estimator used in Monte Carlo GCV. The bounding methods are based on the iterative Lanczos bidiagonalization method. We have created a stopping criteria for the bidiagonalization that stops the iteration when the ratio of upper and lower bounds are within a given tolerance.

The standard-form transformation for Tikhonov regularization used in the previous toolbox and most approaches found in other references were not appropriate in the object oriented framework. A more generally applicable approach has been presented and implemented.

Slow convergence of the BLUR test problem with general-form Tikhonov regularization led to the Tikhonov preconditioner. It demonstrates how a preconditioner is easily incorporated within the object oriented framework. The theoretically computed condition numbers gave a very pessimistic outlook, but experiments showed good results when the problem size grew.

6.4 The Toolbox

The Matlab toolbox M$\mathcal{OO}$Re Tools was created to demonstrate the techniques and ideas of modular algorithms and object oriented programming. Not only is it a proof of concept but it is also a useful tool. It has already been used by several students and graduate students besides the author. It has proven useful as a tool to work with larger problems not possible to solve with the previous Regularization Tools package.

6.5 Further Work

The thesis has described the foundation of the toolbox. The toolbox can already be used for a wide range of problems. However, some parts are more optimized and thoroughly tested than others. The following list points to methods and areas that need more work or would be next in line for fixing if more time was at hand:

- The hybrid methods all minimize the outer problem in a least squares sense. Other penalty functions should be allowed on the outer problem to increase robustness against outliers in the right-hand side.
- The minimization routines used for problems with penalty functions other than the l_2-norm have not been tested and compared to other algorithms. The current optimization algorithm implemented is a Newton type method with line-search. Trust region methods such as Steihaug's CG [119] or the LSTRS [112, 113] method (now also in a non-negativity constrained version [114]) seem promising as an alternative.
- The general LP-solver based on Mehrotra's interior point method which can be used for large-scale problems is not tested well enough. How to precondition the linear systems at each iteration is also a point of further investigation. For small problems we use the `linprog` function from the Optimization Toolbox. The `linprog` is also based on Mehrotra's method, but it cannot use our object oriented framework and we are forced to create the arguments by "de-objecting" the objects into ordinary matrices and vectors and putting the result back into an object.
- Several of the algorithms reduce their problem into a series of least squares problems where a weight matrix is multiplied onto the operator. How do we precondition a large-scale weighted least squares problem? Can we use a good preconditioner to the unweighted least squares problem (such as the Tikhonov preconditioner in §5.3), or do we need another approach. This question is closely related to the previous point as it also can be seen as a weighted least squares problem.
- Parameter choice methods for large-scale problems are not well integrated with the iterative solution methods. Also the parameter choice methods in connection with penalty functions other than l_2-norm penalty have not been investigated. The parameter choice methods for the direct methods use the firm foundation already found in the previous package.
- The standard-form transformation has several formulations. Which specific formulation to use has not been investigated.
- The VectorCollection should implement a QR factorization based on Householder reflections. This extension would enable the use of the "Subspace Preconditioned LSQR" (see §4.6.1 and App. E) and the weighted pseudo-inverse (see §5.2) for a larger group of problems.

- The algorithms in the toolbox are all developed for real arithmetic numbers. It has not been considered which algorithms that at this point work with test problems with complex numbers and which algorithms that need to be altered. A good starting point would be a test problem with a formulation using complex numbers.

Appendices

All this lets infinity in by the back door, while keeping it respectable. It even gets its own symbol, ∞. Infinity lets us do forbidden things, such as divide by zero. When a mathematician writes $1/0 = \infty$, she doesn't mean that 1 divided by 0 makes ∞. She means that if a number x keeps shrinking ever closer to 0, then $1/x$ keeps growing bigger and bigger, without limit. And she has to be a very senior mathematician to be allowed to be that sloppy, even then. Ian Stewart, "Never Ending Story" [120].

The appendix is composed of five parts

1. A tutorial showing how the toolbox is used and how to extend it with new classes.
2. A short overview of the content of the M$\mathcal{OO}$Re Tools toolbox.
3. A listing of the steps taken to test the M$\mathcal{OO}$Re Tools toolbox. Both the classes and the algorithms have been tested.
4. A short introduction to the new test problems introduced in the toolbox.
5. The paper "Subspace Preconditioned LSQR" by Michael A. Saunders, Per Christian Hansen and Michael Jacobsen [71]. It is included as some details are referred to in the main thesis.

Appendix A

Toolbox Tutorial

This appendix shows how to use the M$\mathcal{OO}$Re Tools toolbox. The emphasis is on how the object oriented features change the work flow compared to the old toolbox [68]. Familiarity with Matlab's object oriented features is useful and is described in the Matlab manual [94].

More details on the reasoning around the modular functions can be found in §3.2. The structure of the object oriented framework as well as a case study describing the implementation of the KroneckerProduct2D class can be found in Chapter 4. See also Appendix C where we test the objects and algorithms in the toolbox. It contains many examples on how computations are done with and without the toolbox.

Input to Matlab is written with the usual Matlab prompt >> in front while output from functions are without the prompt. For example,

```
>> 4 + 4
ans =
     8
```

A.1 Installation and Setup

The toolbox comes in a `tar.gz` file that can be downloaded from

`http://www.imm.dtu.dk/nag/software`

It is installed in a UNIX-like system via the following steps (where # is the UNIX prompt):

1. Create a directory for the package.

2. Extract the files to this directory,

   ```
   # cd directory
   # gunzip MOOReTools.tar.gz
   # tar xf MOOReTools.tar
   ```

   On a Windows machine a tool such as "WinZip" can be used to extract the files.

3. Add the directory to the Matlab path. Also the sub-directories TestExamples and Algorithms must be added to the path. The test problems and supporting functions are located in these directories.

4. The package contains a couple of C mex-files which need to be compiled in order for the test problem INTERPOLATE to work. The Matlab script `regmex` compiles the mex-files in the package.

Assuming that the package has been properly set up we continue with the tutorial.

A.2 Creating a Test-Problem

We will start out by creating a test problem with one of the test example generators. We will use the BLUR test problem with images of size 256×256:

```
>> [K,y,x] = blur(256,10,2);
>> whos
Name      Size                  Bytes  Class

K     65536x65536             1050597  KroneckerProduct2D object
y       256x256                524916  Vector2D object
x       256x256                524916  Vector2D object

Grand total is 262179 elements using 2100429 bytes
```

Note that the size of $\mathbf{K}$ is reported as $256^2 \times 256^2$ but the size in bytes is much lower than required for an actual (dense) matrix of that size. Because $\mathbf{K}$ is a KroneckerProduct2D object we can store all necessary information very efficiently and multiplications with Vector2Ds also exploit the Kronecker structure. The vectors are both of class Vector2D. Because we often use Vector2Ds to describe images the class provides a method called show that displays its data as an image. Typing

```
>> show(y)
```

pops up a picture of the Vector2D object `y`, see Fig. A.1.

The KroneckerProduct2D class has "overloaded" a number of operators. We can, for example, multiply with a Vector2D:

```
>> K*x
Vector2D of 256 x 256 = 65536 elements
```

Also Vector2D overloads many operations; we can, for example, take the norm

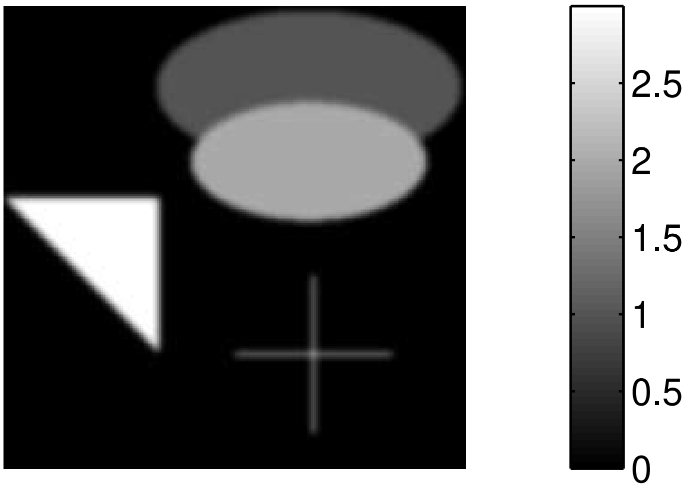


Figure A.1: Blurred image from the BLUR test problem.

```
>> norm(K*x - y)
ans =
     0
>> norm(K\y - x)
ans =
   3.9986e-01
```

where the last command is an example of how "only" having 16 digits of accuracy can have disastrous results. We will now add some noise to `y`

```
>> yn = y + randn(size(y))*1e-4
Vector2D of 256 x 256 = 65536 elements
```

and try some regularization algorithms. Note that Vector2D allows the user to add ordinary two dimensional double arrays with a Vector2D object (as long as number of rows and columns match).

A.2.1 Tikhonov and LSQR

With a KroneckerProduct2D object we can use the SVD version of Tikhonov, because KroneckerProduct2D inherits from SVDOperator and thus it has committed to implementing the SVD. The function `SVDOperator/tikhonov` takes three arguments: the operator, the right hand side and a structure of options

```
>> opt = regset('RegPar', 1e-3);
>> Xtikh = tikhonov(K,yn,opt)
```

The function `regset` checks the option names and whether the values have the right type. If we had been working with a class that did not inherit from SVDOperator it would use the implementation of Tikhonov regularization supplied by LinearOperator. The `LinearOperator/tikhonov` uses LSQR similarly to the way it is shown in the following.

Another option is to use a least squares solver such as LSQR on the augmented system $[\mathbf{K}^T \, \lambda \mathbf{I}]^T$ as done for all LinearOperators that are unable to compute the SVD. For this to work we need the OperatorArray and VectorStack classes which are similar to block matrices and block vectors:

```
>> KL = OperatorArray(2,1);
>> KL{1,1} = K;
>> KL{2,1} = 1e-3*IdentityOperator(256^2);
>> y0 = VectorStack(2);
>> y0{1} = yn; y0{2} = 0*yn;
```

Note how the blocks are indexed with the curly parenthesis `{}`. The IdentityOperator class is an extremely "sparse" representation of the identity, where not even the diagonal of ones is stored. A multiplication is done through a copy of the input—no multiplications with one. The argument to the IdentityOperator constructor is the intended size. The argument is optional but more operations are possible if the IdentityOperator operator knows its size. Now the new objects can be used with LSQR

```
>> opt = regset('Iter', 500, 'TolRes', 1e-2);
>> [Xlsqr,ext] = lsqr_mt(KL,y0,opt);
>> ext.iter
ans =
    88
```

The structure field `ext.iter` tells us that 88 iterations were performed before the tolerance criterion was reached and that the result in `Xlsqr` is the result at that iteration. Most functions return two arguments; the actual result as the first argument and a structure of extra information in the second argument. In the example above, we get the solution and a structure with the field `iter`, that holds the number of iterations actually performed.

A.2.2 Getting Help

The regularization algorithms are located as methods of LinearOperator and SVDOperator. The Tikhonov regularization algorithm exists in two versions,

- one in LinearOperator, that solves the standard-form Tikhonov problem via the iterative LSQR algorithm,
- and one in SVDOPerator, that solves the standard-form Tikhonov problem via the SVD.

To view the online documentation of the two functions use the following lines:

```
>> help LinearOperator/tikhonov
>> help SVDOperator/tikhonov
```

Similarly it is possible to view the documentation of the methods of all other classes. Notice, that Matlab extends the help display with a list of other classes implementing a method having the same name.

A.3 Thesis Examples

In the following we illustrate the toolbox by showing how several of the results in the thesis are computed. Comments on the results are found in the main thesis. This simply illustrates the calling sequences. The final steps of fine tuning the axes, lables etc. of the plots are omitted.

PP-LSQR

See also § 3.4.2 for the mathematical formulation of the PP-LSQR method.

We take the DERIV2 operator, create a piecewise constant solution, a right-hand side and finally we add noise.

```
>> K = deriv2(4000);
>> x = Vector([zeros(1500,1); ones(1500,1); zeros(1000,1)]);
>> y = K*x;
>> noise = randn(4000,1); noise = noise / norm(noise);
>> yn = y + noise*1e-2*norm(y);
```

Then we apply 5 iterations of LSQR and make sure that it returns the Krylov subspace by setting the options `Subspc` to `on`

```
>> lsqropt = regset('Iter', 5, 'Subspc', 'on');
>> [xlsqr, ext] = lsqr_mt(K,yn, lsqropt);
```

Next we compute the modification. We use an approximation to the first derivative provided by the utility function `getL`

```
>> L = getL(4000,1);
>> xperp = modsubs1(L, ext.V, xlsqr);
```

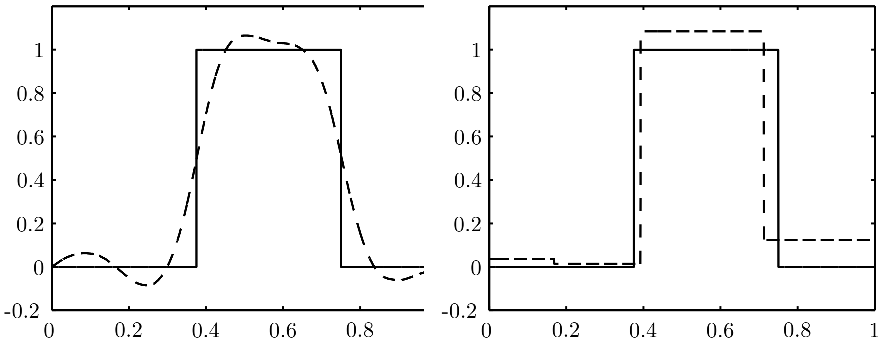
which outputs a line from Optimization Toolbox's `linprog` function telling that the optimization terminated successfully. In Fig. A.2 the unmodified (`xlsqr`) and modified (`xlsqrr+xperp`) solutions are shown along with the true solution.

The results were computed on a Intel Xeon 4 1.8 GHz in approximately 15 seconds, of which the LP-solver used 14 seconds. An SVD of the same operator takes about 2 hours on the same machine.

A GMRES-Tikhonov Hybrid Method

See also § 3.4.3. The test problem is SHAW, created by

```
>> [K,y,x] = shaw(200);
>> noise = randn(size(y)); noise = noise / norm(noise);
>> yn = y + noise*norm(y)*1e-3;
```



(a) Solution (solid) and LSQR (dashed)

(b) Solution (solid) and PP-LSQR (dashed)

Figure A.2: PP-LSQR plots

To compare the hybrid method with a non-hybrid method we do two experiments. In the non-hybrid case we explicitly set the regularization parameter to zero, that is, no inner regularization. In the actual hybrid experiment we use the default GCV parameter choice method. The options to the inner regularization are thus set as follows

```
>> reg = regset('RegPar', 'gcv');
>> noreg = regset('RegPar', 0);
```

We now do 9 iterations with the `ahybrid` method (Arnoldi based, like GMRES). Note how arguments to the inner solver are passed via the `InArgs` option.

```
>> X1 = ahybrid(K, yn, regset('Iter', 9, 'InArgs', noreg));
>> X2 = ahybrid(K, yn, regset('Iter', 9, 'InArgs', reg));
```

Figure A.3 shows the results.

A PP-LSQR-TSVD Hybrid Method

See also § 3.4.5. We now go a step further and combine a hybrid of Lanczos bidiagonalization and TSVD with a modification of the PP type. We create a problem with the HEAT kernel but with the piecewise constant solution (scaled so that the maximum value is 1) from WING:

```
>> n = 400;
>> [K,y,x] = wing(n); x = x / max(x);
>> K = heat(n); y = K*x;
>> noise = randn(size(y)); noise = noise / norm(noise);
>> yn = y + noise*norm(y)*1e-2;
```

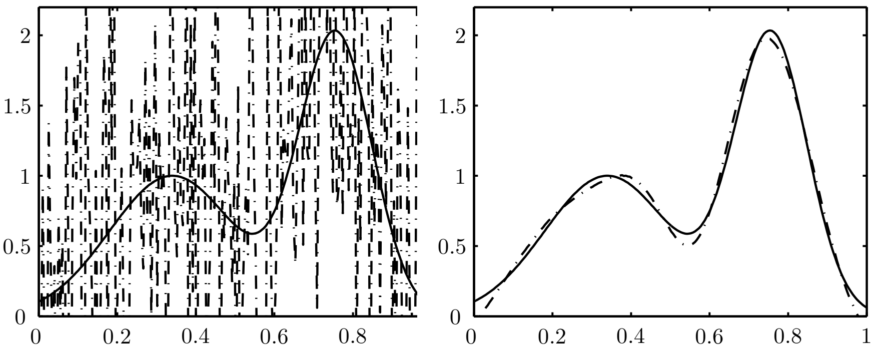


Figure A.3: GMRES and GMRES-Tikhonov

We now setup the options for the inner regularization and for comparison we include an example with no inner regularization, that is, Tikhonov regularization with $\lambda = 0$:

```
>> noregin = regset('RegPar', 0); % For Tikhonov
>> regin = regset('RegPar', 'gcv');
```

The options are passed as options for the inner solvers

```
>> X1 = lbhybrid(K,yn, regset('Iter',iter, 'InSolv','tikhonov', 'InArgs',noregin) );
>> reg = regset('Iter', 7, 'InSolv', 'tsvd', 'InArgs', regin,'Subspc', 'on');
>> [X2, ext] = lbhybrid(K, yn, reg)
```

The `Subspc` option makes `lbhybrid` return the Krylov subspace created during the Lanczos bidiagonalizations, which is needed in the modification step. If we expect a piecewise constant solution, we can achieve a good modification using an approximation to the first derivative created via `getL`. The modification is computed via `modsubs1`:

```
>> L1 = getL(n,1);
>> xp = modsubs1(L1,ext.V, X2)
```

The results, that is, `X1`, `X2` and `X2+xp`, are shown in Fig. A.4.

A Robust Tikhonov Method

See also § 3.4.6. By switching the penalty functions in Tikhonov regularization we are able to create methods that are robust toward outliers in the right hand side. We first create a HEAT test problem and add the usual normal distributed noise, but now we also add 3 outliers:

```
>> n = 300;
>> [K,y,x] = heat(n);
>> randn('state', 0);      % Results are reproducible
>> noise = randn(size(y)); noise = noise/norm(noise);
>> yn = y + noise*norm(y)*1e-2;
```

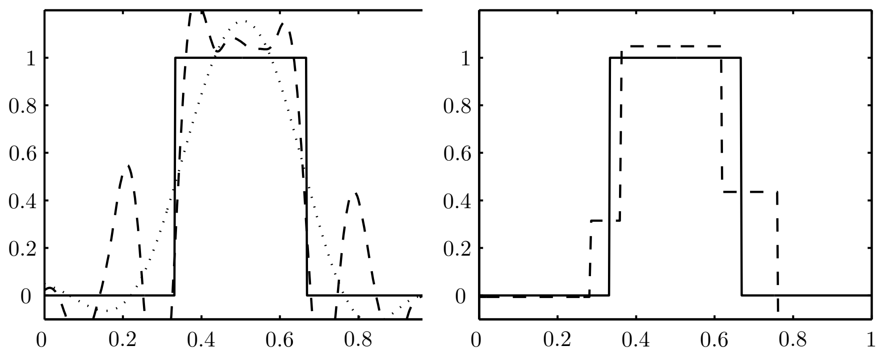


Figure A.4: The LSQR method with inner regularization and modification.

```
>> yno = yn;
>> yno(10) = yn(n/2)*(1 + 0.4);
>> yno(100) = yn(n-30)*(1 - 0.4);
>> yno(200) = yn(190)*(1-0.7);
```

We now create the overdetermined system of equations

```
>> L = 1e-1*getL(n,2);
>> KL   = stack(K,L);
>> yn0  = stack(yn, zeros(L*x));  % Without outliers
>> yno0 = stack(yno, zeros(L*x)); % With    outliers
```

where we have chosen to use an approximation to the second derivative as the regularization matrix.

Then we set up the options structures for `gmin`. We chose to use CGLS and the `inargs` options structure is used by CGLS:

```
>> inargs  = regset('Iter',20);
>> options = regset('Iter',10, 'InSolv', 'cgls', 'InArgs', inargs, ...
                 'Norm', 'stacknorm', ...
                 'NormArgs', {'fair', 0.001 , 1; 'pnorm', 2 , 1});
```

We use the Fair penalty function with weight $\lambda_0 = 1$ and parameter $\beta = 0.001$ for the top block of the overdetermined system of equations $(\mathbf{Kx} - \mathbf{y})$ and the standard l_2-norm squared for the second part $10^{-1}\mathbf{L}_2\mathbf{x}$ also with $\lambda_1 = 1$.

We now solve the problem as a least squares problem and with the alternative combination of norms, and with and without outliers.

```
>> XF = gmin(KL, yno0, options);
>> XFo = gmin(KL, yn0, options);
>> X2o = backslash(KL, yno0);
>> X2 = backslash(KL, yn0);
>> t = linspace(0,1,n);
>> plot(t, x, '-k', t, X2, '--k', t, X2o, ':k');
>> plot(t, x, '-k', t, XF, '--k', t, XFo, ':k');
```

The two plots are shown in Fig. A.5

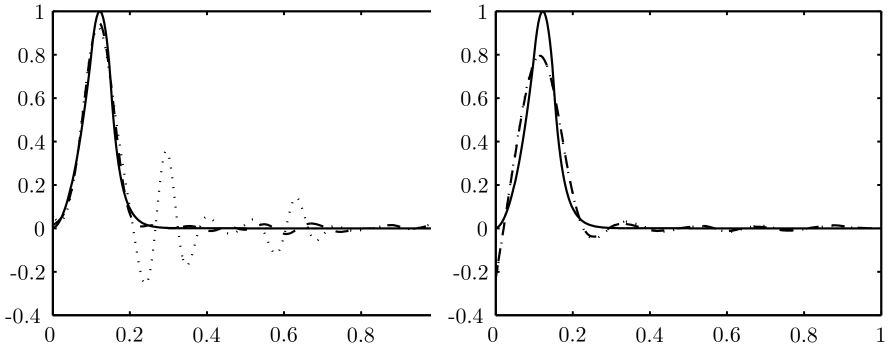


Figure A.5: Robust Tikhonov

Robust Tikhonov with Non-Negativity Constraints

In some cases it can be beneficial to add non-negativity constraints to the solution. The `gmin` has a non-negativity constrained counterpart in `poscg`. Continuing with the problem constructed previously we setup the options structure such that the unconstrained solutions are used as starting guesses

```
>> options = regset(options, 'x0', XF);
>> opt2 = regset(options,'Norm', 'pnorm', 'NormArgs', 2, 'x0', X2o);
>> Xpos   = gminnonneg(KL, yn0, opt2);
>> Xposo  = gminnonneg(KL, yno0, opt2);
>> XFpos  = gminnonneg(KL, yn0, options);
>> XFposo = gminnonneg(KL, yno0, options);
>> plot(t, x, '-k', t, Xpos, '--k', t, Xposo, ':k');
>> plot(t, x, '-k', t, XFpos, '--k', t, XFposo, ':k');
```

Note how we in the second line is able to update fields in our option structure. The result is shown in Fig. A.6.

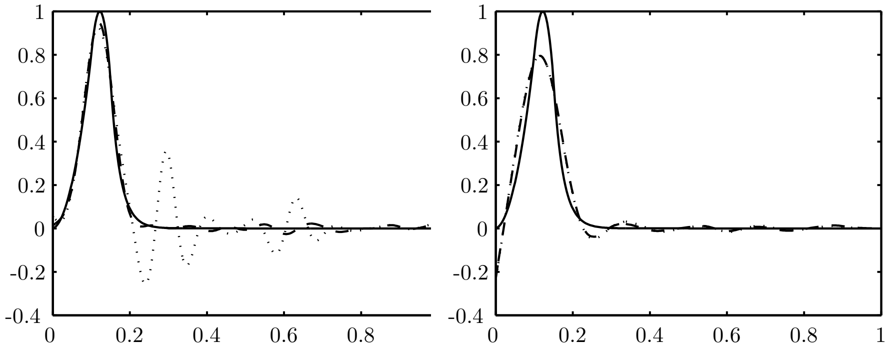


Figure A.6: Non-Negativity robust Tikhonov

Using the Weighted Pseudo-Inverse

See also § 5.2. The weighted pseudo-inverse is used to transform a general-form Tikhonov problem into a standard-form Tikhonov problem. We will use the standard-form transformation in connection with a parameter choice method. In this case we will use the bounding of Regińska's L-curve.

We first setup the problem. We will use a Kronecker product of two DERIV2 test matrices and the stacked regularization matrix

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_1 \otimes \mathbf{I} \\ \mathbf{I} \otimes \mathbf{L}_1 \end{bmatrix}.$$

We will use the standard solution provided by BLUR. The setup can be done as follows:

```
>> n = 40;
>> [K,y,x] = blur(n);
>> K = deriv2(n);
>> K = kron(K,K);
>> y = K*x;
>> [L1,W1] = getL(n,1);
>> I = IdentityOperator(n);
>> L = stack(kron(L1,I), kron(I,L1));
>> W = kron(W1, W1);
```

Observe that `kron` is overloaded by LinearOperator to yield a KroneckerProduct2D object as result. That is `kron(L1,I)` is equivalent to `KroneckerProduct2D(L1,I)`. We now add normally distributed noise to the right-hand side:

```
>> randn('state', 0); % We can reproduce results
>> noise = randn(size(y));
>> noise = noise / norm(noise, 'fro'); % Normalize in 2D
>> yn = y + noise*norm(y)*1e-2;
```

We set up the weighted pseudo-inverse via its constructor that takes the regularization matrix $\mathbf{L}$, its null-space $\mathbf{W}$ and the operator $\mathbf{K}$ as arguments. We also find the solution in the null-space as well as the transformed right-hand side:

```
>> LKinv = WeightedPseudoInverse(L,W,K);
>> x0 = nullcomp(LKinv, yn);
>> yhat = yn - K*x0;
```

Finally, we use the bounding method for Regińska's L-curve on the transformed problem with two different stopping criterion tolerances:

```
>> opt = regset('RegPar', [1e-6 1e-1], 'Disp', 1, 'Iter', 200);
>> [reg1, ext1] = reginska(K*LKinv, yhat, regset(opt, 'TolRes', 2));
>> [reg2, ext2] = reginska(K*LKinv, yhat, regset(opt, 'TolRes', 0.1 ));
```

The computation is not "instant" as several least squares problems with $\mathbf{L}$ and $\mathbf{L}^T$ need to be solved.

The Tikhonov Preconditioner

The Tikhonov preconditioner is described in Sec. 5.3. It was created to accelerate convergence of the BLUR test problem with non-identity regularization operator.

We first create a problem that we cannot solve by direct methods:

```
>> n = 512;
>> lambda = 1e-5;
>> mu = 1e-4;
>> iter = 200;
>> noise = 1e-5;
>> [K,y,x] = blur(n,8);
>> randn('state',0); % Reproduceable results
>> noise = randn(size(b));
>> ys = y + noise/norm(noise)*norm(y)*1e-4;
>> L1 = getL(n,1);
>> L0 = getL(n,0);
>> L  = stack(KroneckerProduct2D(L1,L0), ...
              KroneckerProduct2D(L0,L1));
>> y0 = L*stack(x);
```

We use the `stack` function to quickly create an OperatorArray object and a VectorStack object used for the least squares formulation of the general-form Tikhonov problem:

```
>> KL = stack(stack(K),lambda*L);
>> y0 = stack(stack(bs),zeros(y0));
```

We try to do `iter` iterations and storing the result after each iteration with

```
[X1,ext] = cgls(KL,b0,regset('Iter', 1:iter, 'TolRes',1e-6));
```

Creating the preconditioner is as simple as providing the operator and a regularization parameter to the constructors:

```
>> P = TikhPrecond(K, mu);
```

Using it is as simple as adding it to the list of options for CGLS

```
>> opt = regset('Iter', 1:iter, 'TolRes', eps/10, 'Precond', P));
>> [X2, ext2] = cgls(KL,b0,opt)
```

We set the relative residual tolerance to one tenth of the machine precision (eps) to ensure high precision.

A.3.1 Writing a new LinearOperator child

It can be complicated and it takes some effort to implement a new LinearOperator child. Basically two classes of operators exist—one more capable than the other. The simplest operator type only implements multiplication with a vector and perhaps multiplication of the transposed operator with a vector. A more advanced operator also implements one or more factorizations. Assume that the new class is called NewOperator

First create a directory named `@NewOperator` and create a file `NewOperator.m` for the constructor in this directory. A template looks like

```
function A = NewOperator(arg)

A.data = arg;
parent = LinearOperator;
A = class(A, 'NewOperator', parent);
```

The last line puts NewOperator into the class hierarchy with LinearOperator as its parent. If a function is called with our new operator as an argument Matlab will first search the directory `@NewOperator` for a m-file (called a method) of that name. If it is not found it continues searching the parent's directory—in this case `@LinearOperator`.

The maybe most important operation for NewOperator is to apply itself to a vector. The parent LinearOperator helps with the implementation by overloading `mtimes`, that is, the multiplication operator `*`. The `@LinearOperator/mtimes` method assists by handling details of multiplications with scalars and NewOperator only needs to consider how to apply itself to a vector. This should be done by the method `@NewOperator/sub_applytovector.m`. When an object inheriting from LinearOperator is multiplied with a Vector the function `@LinearOperator/mtimes.m` is called. It calls the method `sub_applytovector.m` function to do the actual work and afterward it scales the result, that is, `1e-2*K` does not modify elements of `K` but merely sets a scaling variable in the operator. The `sub_applytovector.m` function must check whether the operator is transposed. A generic template is

```
function y = sub_applytovector(K, x)
if gettransposed(K)
  % Calculate y = K'*x or error if not possible
else
  % Calculate y = K*x
  y = fun(K.data, getvector(x))
end
```

where the appropriate code must be inserted. See also § 4.3.1. Actual code should also include tests of parameter types etc.

Also keep in mind that the result of the multiplication is an object in the Vector hierarchy of the proper type. For instance, the result of applying a KroneckerProduct2D to a Vector2D object should give a Vector2D object. In other cases the classes used for of "domain" and "range" vectors might be different.

For further examples see the already implemented classes. The class KroneckerProduct2D is an example of a class that implements both multiplication and several factorizations. The class Interpolate in the test examples only implements multiplications (through mex-files).

A.3.2 Writing a new Vector child

Writing a new Vector descendant is easier than writing a LinearOperator class. The Vector2D inherits almost all of its functionality from the base class Vector. The base class Vector implements plus, minus, less than etc. using the corresponding operators on the output from the method `getdata`. The result is then stored via the method `setdata`. This means that if the following is true

```
z = setdata(x, getdata(x) + getdata(y))
```

in the case of addition the only thing needed for the new Vector descendant is to implement the constructor and the two methods `setdata` and `getdata`. The Vector2D does that as most operators such as + work elementwise.

Finally the NewVector must also implement the inner product of two NewVectors in its method `inner` and the `norm` method.

However, one of the useful aspects of defining a new operator is the possiblity of implementing a proper plotting method. E.g. the Vector2D has a method `show` that displays the data as a graytone image with a colorbar. Other possiblities is that NewVector also contains information on the particular discretization used that could be used both for plotting and for operations in connection with a NewOperator that adapts to the size of the input vector. An example is the GridVector used by the Interpolate operator.

A.3.3 Using the Option Structure

Inspired by the Optimization Toolbox our toolbox uses a structure to pass options to the methods. Like `optimget` and `optimset` from the Optimization Toolbox the toolbox provides the functions `regget` and `regset` to simplify the passing and handling of options.

The function `regset` sets fields of a structure but first it checks whether data has a valid type. As an example the field `'Iter'` is only allowed to have doubles as argument. The `'Iter'` field can be set by one of the following lines

```
opt = regset( 'Iter', 10 )
opt = regset( oldopt, 'Iter', 10)
```

where the second line adds or changes the `'Iter'` field of `oldopt` and returns the updated option structure.

The function `regget` retrieves a field from an options structure. If the field is empty it can return a default value and thereby it significantly reduces the complexity of the initial stages of a function. An example is

```
iter = regget(options, 'Iter', 10)
```

where `regget` retrieves the option `'Iter'` in the `options` structure. If the field is empty `regget` returns 10.

Table A.1 shows the options that can be used at the time of writing. See the help of a method to see precisely which options it uses.

If the predefined set of options is insufficient a new field can be added by taking the following steps:

1. In `regset` add the new field to the structure `options`.
2. Add the name of the field to the type checking switch statement in the `checkfield` sub-function at the end of `regset` file.
3. In `regget` add the new field to the structure `optionsstruct`.

Option name	Description
Disp	Verbosity level
InArgs	Arguments to an inner solver
InSolv	Name of an inner solver
Iter	Number of outer iterations
NormArgs	Arguments to a norm-function
Norm	Name of a norm-function
Precond	Preconditioner for outer solver
RegPar	Regularization parameter for method
Reorth	Reorthogonalization of e.g. Lanczos vectors
Subspc	Return (e.g. Lanczos) subspace
TolRes	Residual tolerance
TolCon	Constraint tolerance
TolX	Tolerance for solution
v1	Start vector for hybrid methods
x0	Starting guess
x_true	True solution

Table A.1: Fields in the options structure. The actual purpose of each option depends on the particular function.

Appendix B

M$\mathcal{OO}$Re Tools Contents

The M$\mathcal{OO}$Re Tools package includes a number of regularization methods. The iterative regularization methods are placed as methods in the LinearOperator class, regularization methods that use the SVD are placed in the SVDOperator class. The Tikhonov regularization method is special as it exist in both LinearOperator (where it uses LSQR) and SVDOperator (where it uses the SVD). That is, the SVD is used to compute the Tikhonov solution when the SVD is available. Otherwise we use LSQR on the damped least squares problem. Table B.1 lists the regularization methods that are available in M$\mathcal{OO}$Re Tools at the time of writing. Table B.2 shows the parameter choice methods and Table B.3 lists the functions that are not directly regularization methods but used as tools or modules in the actual regularization methods. See the online help, App. A and App. C for further details and examples on how the functions are used.

Because most functions are placed as members of the LinearOperator and SVDOperator classes they can only be invoked if they are called with arguments of those types. For example to invoke `lsqr_mt` we need `K` to be a LinearOperator object

```
>> K = Matrix(randn(8)); x = Vector(randn(8,1)); y = K*x;
>> xlsqr = lsqr_mt(K,y, regset('Iter',3));
```

To obtain help use the usual `help` command, e.g.,

```
>> help lsqr_mt
```

In cases where a method exists in two classes, such as `tikhonov`, we need to specify the class also. For instance,

```
>> help LinearOperator/tikhonov
>> help SVDOperator/tikhonov
```

	Loc.	Name	Short description
	L	`ahybrid`	Hybrid method based on the Arnoldi (gmres) method
	L	`cg`	Conjugate gradient method
*	L	`cgls`	Conjugate gradient least squares method
	A	`glsqr`	LSQR solution with general modification
	L	`gmin`	A minimization method for penalty type regularization
	L	`gminnonneg`	A minimization method for penalty type regularization with non-negativity constraint.
	L	`gmres_mt`	The GMRES method
	L	`l1sol`	Solve constrained l_1 problem
	S	`l1tsvd`	TSVD in 1-norm
	S	`lptsvd`	TSVD in p-norm
	L	`lbhybrid`	Hybrid method with Lanczos bidiagonalization
*	L	`lsqr_mt`	The LSQR method
	L	`lthybrid`	Hybrid method with Lanczos tridiagonilizaton
	L	`minres_mt`	The MINRES method
	A	`mlsqr`	LSQR with modification
*	A	`mtsvd`	TSVD with modification
	L	`mrII`	The MINRES method II
	A	`ppgmres`	GMRES with "PP" modification
	A	`pplsqr`	LSQR with "PP" modification
*	LS	`tikhonov`	Tikhonov regularization
*	S	`tsvd`	TSVD in 2-norm (the usual)

Table B.1: Regularization functions. The Loc. column indicates where to find the function; L indicates that is a member of LinearOperator, S tells that it is a method of SVDOperator and A that the function is found in the Algorithm directory. LS implies that two implementations exist, one in LinearOperator and one in SVDOperator. The functions marked with "*" are also found in the previous package Regularization Tools.
A penalty method denotes a method, such as Tikhonov, where we add a penalizing term to the problem, see § 3.1.1 for more details.
A hybrid method denotes a method which combines a projection method, such as LSQR, and regularization on the projected problem, see § 3.2.2.
A modification combines the solution from a projection method with a modification found in the subspace orthogonal to the projection. E.g., a "PP" modification implies adding the solution to a minimization of $\|\mathbf{Lx}\|_1$ subject to an orthogonality constraint. See also Table B.4 and § 3.2.3.

	Loc.	Name	Short description
*	S	`gcv`	Generalized cross validation
	L	`gcvbounds`	MC generalized cross validation bounds
*	S	`l_curve`	Plot L-curve and locate corner
	L	`lcurvebounds`	Bounds for L-curve curvature
	L	`reginska`	Reginska L-curve bounds

Table B.2: Parameter choice methods. See Table B.1 for explanation of the Loc. column. The functions marked with "*" are also present in Regularization Tools.

Loc.	Name	Short description
A	`gmodsubs`	General minimization with orth. constraint
L	`lanczos`	Lanczos tridiagonalization
A	`l1orth`	1-norm minimization with orth. constraint
A	`linesearch`	Line search function
A	`lsorth`	2-norm minimization with orth. constraint
A	`modsubs`	Compute 2-norm modification (uses lsorth)
A	`modsubs1`	Compute 1-norm modification (uses l1orth)
A	`regget`	Option structure management (getting)
A	`regset`	Option structure management (setting)

Table B.3: Supporting methods. See Table B.1 for an explanation of the Loc. column.

	`modsubs1`	`gmodsubs`	`modsubs`
`ahybrid`			
`cg`			
`cgls`			
`gmres`	`ppgmres`		
`l1tsvd`			
`lptsvd`			
`lbhybrid`			
`lsqr`	`pplsqr`	`glsqr`	`mlsqr`
`lthybrid`			
`minres`			
`mrII`			
`tsvd`			`mtsvd`

Table B.4: Combination of modules. The first column shows the modules that find a solution "in a subspace" while the first row shows the modules that find the "modifications". Inside the matrix we indicate the combinations that are implemented. The table is sparse but all other combinations are easily created using one of the already implemented functions as templates.

Appendix C

Validation of Toolbox

This appendix describes the procedure (that is, the scripts) used to test the classes and algorithms contained in M$\mathcal{OO}$Re Tools.

Several of the tests compare results from M$\mathcal{OO}$Re Tools with results obtained using the Regularization Tools [67] package. Hence, Regularization Tools must be available in the Matlab path. Furthermore we use the function `l1c` which is embedded as a subfunction in `pptsvd.m` available under "Additional Matlab Software" at

`http://www.imm.dtu.dk/~pch/Regutools/index.html`.

The embedded subfunction `l1c` must be extracted into its own "m-file".

The Optimization Toolbox is in several cases used for comparison and is therefore also needed to run all tests. The test functions are included in the M$\mathcal{OO}$Re Tools tar-file in the TestScripts directory.

An "über-script" `testall` is created that runs all the scripts/functions described in the following:

```
% Run all test scripts

disp('------*  Testing Objects  *-------');
disp('Testing Vector');                 testvector
disp('Testing Vector2D');               testvector2d
disp('Testing Vector3D');               testvector3d
disp('Testing VectorND');               testvectornd
disp('Testing Matrix');                 testmatrix
disp('Testing SparseMatrix');           testsparse
disp('Testing Identity');               testidentity
disp('Testing Diagonal');               testdiagonal
disp('Testing KroneckerProduct2D');     testkron2d
```

```
disp('Testing KroneckerProduct3D');    testkron3d
disp('Testing KroneckerProduct');      testkron
disp('Testing NullOperator');          testnulloperator
disp('Testing OperatorProduct');       testoperproduct
disp('Testing OperatorArray');         testoperarray
disp('Testing OperatorSum');           testopersum
disp('Testing PermutationOperator');   testpermutation
disp('Testing VectorCollection');      testvectorcollection
disp('Testing VectorReshape');         testvectorreshape
disp('Testing TikhPrecond');           testtikhprecond
disp('Testing WeightedPseudoInverse'); testweightedpseudoinverse
disp('Testing OperatorSVD');           testoperatorsvd

disp(' ');
disp('------*  Testing Algorithms   *-----');
disp('Testing ahybrid');               testahybrid
disp('Testing cg');                    testcg
disp('Testing cgls');                  testcgls
disp('Testing gmin');                  testgmin
disp('Testing gminnonneg');            testgminnonneg
disp('Testing gmres');                 testgmres
disp('Testing l1orth');                testl1orth
disp('Testing l1sol');                 testl1sol
disp('Testing l1tsvd');                testl1tsvd
disp('Testing lbhybrid');              testlbhybrid
disp('Testing lptsvd');                testlptsvd
disp('Testing lsqr');                  testlsqr
disp('Testing lthybrid');              testlthybrid
disp('Testing mrII');                  testmrii
disp('Testing minres');                testminres
disp('Testing tikhonov (svd)');        testtikhonovsvd
disp('Testing tikhonov (lo)');         testtikhonovlo
disp('Testing tsvd');                  testtsvd
disp('Testing gmodsubs');              testgmodsubs
disp('Testing lsorth');                testlsorth
disp('Testing modsubs');               testmodsubs
disp('Testing modsubs1');              testmodsubs1

disp(' ');
disp('------* Testing parameter choice *------');
disp('Testing gcv');                   testgcv
disp('Testing l_curve');               testlcurve
disp('Testing reginska');              testreginska
disp('Testing gcvbounds');             testgcvbounds
disp('Testing lcurvebounds');          testlcurvebounds

disp(' ');
disp('------* Testing combined algorithms *------');
disp('Testing glsqr');                 testglsqr
disp('Testing mlsqr');                 testmlsqr
disp('Testing mtsvd');                 testmtsvd
disp('Testing ppgmres');               testppgmres
disp('Testing pplsqr');                testpplsqr
```

C.1 Testing the **Vector** hierarchy

The Vector class, and the classes inheriting from it, all implement the same interface. Therefore we construct a function that calls the functions defined by the common interface. For each class we then add code to test special code of the particular class. The `getvector` method is used in the tests of all other functions and thus we test the `getvector` method first. The following function checks the common methods of the Vector interface:

```
function testfunvector(v1,v2,vv1,vv2)
% Test the functions of a vector
%
% Two arguments to allow for tests of dyadic operators

v2s = {
    'all';     'any';
    'max';     'min';
    'norm';    'sum'    };

v2v = {
    'abs';     'cosh';
    'ctranspose';
    'exp';     'log';
    'not';     'sign';
    'sqrt';     'tanh';
    'uminus';   'uplus';
    'isinf';   'conj' };

vs2v = {
    'and';     'eq';
    'ge';       'gt';
    'le';       'lt';
    'minus';   'ne';
    'or';       'plus';
    'power';   'rdivide';
    'ldivide'; 'times';
    'xor'
    };

sv2v = {
    'and';      'eq';
    'ge';       'gt';
    'le';       'lt';
    'minus';   'ne';
    'or';       'plus';
    'rdivide'; 'ldivide';
    'times';   'xor'
    };

vv2v = {
    'and';     'eq';
    'ge';     'gt';
    'le';     'lt';
    'minus';  'ne';
```

```
    'or';      'plus';
    'power';  'rdivide';
    'ldivide';'times';
    'xor'
    };

checkgetvector(v1,vv1);
checkgetvector(v2,vv2);

for i=1:length(v2s)
  checkv2s(v2s{i}, v1)
  checkv2s(v2s{i}, v2)
end

for i=1:length(v2v)
  checkv2v(v2v{i}, v1);
  checkv2v(v2v{i}, v2);
end

for i=1:length(vs2v)
  checkvs2v(vs2v{i}, v1, 2);
  checkvs2v(vs2v{i}, v2, 0.1);
end

for i=1:length(sv2v)
  checksv2v(sv2v{i}, 2   ,v1);
  checksv2v(sv2v{i}, 0.1 ,v2);
end

vt = v1';
if ~gettransposed(vt)
  printerror('transposing', v1);
end

n1 = norm(v1)^2; n2 = norm(v2)^2;
if abs(v1'*v1 - n1) > length(v1)*eps,
  printerror('inner', v1, abs(v1'*v1 - n1));
end
if abs(v2'*v2 - n2) > length(v2)*eps,
  printerror('inner', v2, abs(v2'*v2 - n2));
end

%-----
function checkgetvector(v,vv)

d = norm(getvector(v)-vv);
if d > length(vv)*eps
  printerror('getvector', v, d);
end

%-----
function checkvs2v(fun, v, s)

VV = feval(fun , getvector(v), s);
vv = getvector(feval(fun , v , s));
```

```
if ~isequal(VV,vv)
  printerror(fun, v, norm(VV-vv));
end

%-----
function checksv2v(fun, s, v)

VV = feval(fun , s, getvector(v)  );
vv = getvector(feval(fun , s , v) );

if ~isequal(VV,vv)
  printerror(fun, v, norm(VV-vv) );
end

%-----
function checkv2s(fun ,v)

S = feval(fun, getvector(v));
s = feval(fun, v);

if ~isequal(S,s)
  printerror(fun, v, abs(s-S));
end

%-----
function checkv2v(fun, v)

VV = feval(fun, getvector(v));
vv = getvector(feval(fun, v));

if ~isequal(VV,vv)
  printerror(fun, v, norm(VV-vv));
end
```

The checks are performed by calling the functions using objects and comparing the results with the corresponding ones computed with ordinary Matlab double arrays. The `printerror` function prints the name of the method, the class name and optionally a scalar. The scalar is used in cases where the result of a computation can vary depending on the exact order of, for example, a summation. The value is used to determine if we have found an error or the different results are a consequence of rounding errors due to another order of computations. Thus the output must be inspected to determine if any unusually large discrepancies have occurred. Note, that a "large discrepancy" is relative to the particular function that we are testing.

C.1.1 Test of Vector

The Vector class is tested by creating two objects and using the function described above. We also test if the constructor reports errors for non-column vector arguments as it should:

```
function testvector()
```

```
vv1 = randn(8,1); vv2 = randn(8,1);
v1 = Vector(vv1); v2 = Vector(vv2);
testfunvector(v1,v2,vv1,vv2)

testerror('Vector(randn(9,3))')
```

Test of Vector2D, Vector3D and VectorND

The test of Vector2D, Vector3D closely follows the steps used to test the base class Vector.
Test of Vector2D:

```
function testvector2d()

vv1 = randn(8); vv2 = randn(8);
v1 = Vector2D(vv1); v2 = Vector2D(vv2);
testfunvector(v1,v2,vv1(:),vv2(:));
testerror('Vector2D(randn(9,2,4))');
```

Test of Vector3D:

```
function testvector3d()

vv1 = randn(8,8,3); vv2 = randn(8,8,3);
v1 = Vector3D(vv1); v2 = Vector3D(vv2);
testfunvector(v1,v2,vv1(:),vv2(:));
testerror('Vector3D(randn(8,2,9,3))');
```

Test of VectorND:

```
function testvectornd()

vv1 = randn(8,8,3,4); vv2 = randn(8,8,3,4);
v1 = VectorND(vv1); v2 = VectorND(vv2);
testfunvector(v1,v2,vv1(:),vv2(:))
```

Test of VectorStack

The following script tests the various parts of the VectorStack class. The creation of the VectorStack objects is somewhat more complicated than for the simpler classes as it needs other Vectors to be created. See also the test of the OperatorArray class where we test multiplication of OperatorArray and VectorStack objects.
Test of VectorStack:

```
function testvectorstack();

vv11 = randn(8,1);   vv21 = randn(8,1);
vv12 = randn(8);     vv22 = randn(8);
vv13 = randn(8,8,2); vv23 = randn(8,8,2);
v11 = Vector(vv11);   v21 = Vector(vv21);
v12 = Vector2D(vv12); v22 = Vector2D(vv22);
v13 = Vector3D(vv13); v23 = Vector3D(vv23);
```

```
v1 = stack(v11,v12,v13); vv1 = [vv11(:); vv12(:); vv13(:)];
v2 = stack(v21,v22,v23); vv2 = [vv21(:); vv22(:); vv23(:)];

testfunvector(v1,v2,vv1,vv2)
```

C.2 Testing the LinearOperator hierarchy

The tests of LinearOperator type objects have one or more phases:

1. A test of the interface defined by LinearOperator. This test includes testing of multiplication and solution methods, if applicable.
2. A test of the interface defined by SVDOperator, if applicable. This part consist of testing the `svd` operation.
3. A test of the interface defined by QROperator, if applicable.

The base classes LinearOperator, SVDOperator and QROperator are tested indirectly through the tests of the derived classes.

LinearOperator Interface Tests

The following scripts are used to test the different interfaces defined in LinearOperator. They are used in the tests of the operators in the LinearOperator hierarchy.

Of vital importance is the test of the `getmatrix` method because it is the foundation of all other tests. We compare the result of `getmatrix` with a double array constructed with Matlab's usual commands:

```
function testgetmatrix(A,AA)

d = norm(getmatrix(A) - AA, 'fro');
if d > prod(size(AA))*eps*10
  printerror('getmatrix', A, d);
end
```

The `testsize` function checks if the size is reported correctly by the object:

```
function testsize(A)

mn = size(A); mmnn = size(getmatrix(A));
if ~isequal(mn,mmnn),  printerror('size', A); end
```

The `testmtimes` function checks if multiplication with an object yields the same result as multiplication with the object converted to a double array. This function checks operator-vector multiplication:

```
function testmtimes(A,x)

AA = getmatrix(A);
b  = getvector(A*x); bb = AA*getvector(x);
d = norm(b - bb);
if d > eps*length(x),  printerror('mtimes', A, d); end
```

Some operators also support multiplication with operators. The `testopermult` function checks if multiplication of two operators yields the proper result. The third argument is an object of the class that the result of the multiplication is supposed to be:

```
function testopermult(A,B,C)

AA = getmatrix(A); BB = getmatrix(B);
AB = A*B;

if ~isequal(class(AB), class(C)),  printerror('opermult',A); end

d = norm(getmatrix(AB) - AA*BB, 'fro');
if d > eps*prod(size(A))*10
  printerror('Not exact match in operator multiplication', A, d);
end
```

The `testplusminus` function performs a similar test but now addition and subtraction of operators are tested:

```
function testplusminus(A,B,C)

APB = A+B; AMB = A-B;

if ~isequal(class(APB) , class(C)), printerror('plus',A);  end
if ~isequal(class(AMB) , class(C)), printerror('minus',A); end

d = norm(getmatrix(A+B)-(getmatrix(A)+getmatrix(B)),'fro');
if d > eps*prod(size(A))
  printerror('plus', A, d);
end
d = norm(getmatrix(A-B)-(getmatrix(A)-getmatrix(B)),'fro');
if d > eps*prod(size(A))
  printerror('minus', A, d);
end
```

The `testsolve` function checks if the solve operation yields a result close to the result obtained via Matlab's `pinv`:

```
function testsolve(A,y)

AA = getmatrix(A); yy = getvector(y);
xx = pinv(full(AA))*yy;
x  = getvector(A\y);

d = norm(xx - x);
if d > eps*length(x),  printerror('sub_solve', A, d); end
```

The function `testdiag` tests if the diagonal is returned as expected:

```
function testdiag(A)

d = norm(getvector(diag(A)) - diag(getmatrix(A)));
if d > max(size(A))*eps,  printerror('diag', A, d); end
```

The function `testcond` tests the condition number computation:

```
function testcond(A)

d = abs(cond(A)-cond(full(getmatrix(A))));
if d > eps, printerror('cond', A, d); end
```

In all tests seen later we scale and transpose the operators to see if the interplay between LinearOperator's `mtimes` and the particular object's `sub_applytovector` works correctly.

SVDOperator Interface Test

The SVD is unique up to a change of sign in the singular vectors (assuming that none of the singular values are equal). We check if the singular vectors from the object oriented framework are orthogonal to the SVD vectors obtained from the operator's double array. Also the singular values are checked to see if they match:

```
function testsvd(A)

[U,S,V]    = svd(A);
U = getmatrix(U);S = getmatrix(S);V = getmatrix(V);
[UU,SS,VV] = svd(full(getmatrix(A)));

p = length(find(diag(SS) < SS(1,1)*eps*10 ));
d = norm(UU(:,1:p)'*U(:,1:p) - VV(:,1:p)'*V(:,1:p), 'fro');
if d > eps*size(U,1),  printerror('svd vectors', A, d); end

d = norm(diag(S) - diag(SS));
if d > eps*min(size(S)),  printerror('svd values', A, d); end
```

QROperator Interface Test

The QR factorization is not unique. Thus we check if we have an orthogonal matrix and a triangular matrix which, when multiplied, gives the input operator:

```
function testqr(A,varargin)

[Q,R] = qr(A,varargin{:});
QQ = getmatrix(Q); RR = getmatrix(R);

d = norm(QQ'*QQ - eye(size(Q,2)), 'fro');
if d > eps*size(Q,1),  printerror('qr vectors', A, d);end

d = norm(RR-triu(RR),'fro');
if d > eps*prod(size(R)),  printerror('qr tri', A, d);end

d = norm(QQ*RR - getmatrix(A), 'fro');
if d > eps*prod(size(A))*10, printerror('qr prod', A, d); end
```

C.2.1 Test of DiagonalOperator

The simple DiagonalOperator supports a wide range of operations due to its simple nature. We use both Vector and Vector2D objects to describe the diagonal:

```
function testdiagonal()

d = randn(64,1);
AA = diag(d); A = DiagonalOperator(Vector(d));
x = Vector(randn(64,1));

test1(A   ,x,x,  AA);
test1(2*A ,x,x,2*AA);
test1(3*A',x,x,3*AA');

AA = full(spdiags(d, 0, 64, 100));
A  = DiagonalOperator(Vector(d), 64, 100);
x  = Vector(randn(100,1)); y = Vector(randn(64,1));

test1(  A ,x,y,AA);
test1(2*A ,x,y,2*AA);
test1(3*A',y,x,3*AA');

d1 = randn(64,1); d2 = randn(64,1);
D1 = DiagonalOperator(Vector(d1));
D2 = DiagonalOperator(Vector(d2));
I  = IdentityOperator(64);

testopermult(  D1,D2 , D1);
testopermult(2*D1,D2', D1);
testopermult(2*D1',I , D1);

d = randn(20);
AA = diag(d(:)); A = DiagonalOperator(Vector2D(d));
x = Vector2D(randn(20));

test1(A   ,x,x,  AA);
test1(2*A ,x,x,2*AA);
test1(3*A',x,x,3*AA');

%-----
function test1(A,x,y,AA)
testgetmatrix(A,AA);
testmtimes(A,x);
testcond(A);
testsvd(A);
testqr(A);
testsolve(A,y);
testsize(A);
testdiag(A);
testplusminus(A,A,A);
```

C.2.2 Test of **IdentityOperator**

The test of DiagonalOperator is similar to the test of DiagonalOperator. Note that the result of multiplication of two IdentityOperators is a DiagonalOperator:

```
function testidentity()

AA = eye(64); A = IdentityOperator(64);
```

```
x = Vector(randn(64,1));

test1(A   ,x,x,  AA);
test1(2*A ,x,x,2*AA);
test1(3*A',x,x,3*AA');

AA = eye(64, 100);
A  = IdentityOperator(64, 100);
x  = Vector(randn(100,1)); y = Vector(randn(64,1));

test1(  A ,x,y,AA);
test1(2*A ,x,y,2*AA);
test1(3*A',y,x,3*AA');

% ---
function test1(A,x,y,AA)
testgetmatrix(A,AA);
testmtimes(A,x);
testsvd(A);
testqr(A);
testcond(A);
testsolve(A,y);
testsize(A);
testdiag(A);
testplusminus(A,A,DiagonalOperator);
```

C.2.3 Test of **KroneckerProduct**

The test of the KroneckerProduct class includes a test of multiplication between two four-term KroneckerProduct objects:

```
function testkron()

disp('Check 1 - square terms')
AA = randn(4); BB = randn(4); CC = randn(4); DD = randn(4);
xx = randn(4,4,4,4); yy = randn(4,4,4,4);
A = Matrix(AA); B = Matrix(BB); C = Matrix(CC); D = Matrix(DD);
x = VectorND(xx); y = VectorND(yy);

ABC1 = KroneckerProduct({A,B,C,D});
AABBCC1 = kron(AA,kron(BB,kron(CC,DD)));
test1(ABC1,x,y,AABBCC1)

disp('Check 2 - all overdetermined ')
AA = randn(4,3); BB = randn(4,3); CC = randn(4,3); DD = randn(4,3);
xx = randn(3,3,3,3); yy = randn(4,4,4,4);
A = Matrix(AA); B = Matrix(BB); C = Matrix(CC); D = Matrix(DD);
x = VectorND(xx); y = VectorND(yy);

AABBCC2 = kron(kron(AA,BB),kron(CC,DD));
ABC2 = KroneckerProduct({A,B,C,D});
test1(ABC2,x,y,AABBCC2)

disp('Check 3 - mixed (problems in solve)')
AA = randn(4,4); BB = randn(3,7); CC = randn(4); DD = randn(3);
```

```
xx = randn(3,4,7,4); yy = randn(3,4,3,4);
A = Matrix(AA); B = Matrix(BB); C = Matrix(CC); D = Matrix(DD);
x = VectorND(xx); y = VectorND(yy);

AABBCC3 = kron(AA,kron(BB,kron(CC,DD)));
ABC3 = KroneckerProduct({A,B,C,D});
test2(ABC3,x,y,AABBCC3)

disp('Test 4 - multiplication');
testopermult(ABC1,ABC2,ABC1);

% -----
function test1(K,x,y,KK)
testgetmatrix(K,KK);
testmtimes(K,x);
testcond(K);
testsvd(K);
testqr(K,0);
testsolve(K,y);
return

% -----
function test2(K,x,y,KK)
testgetmatrix(K,KK);
testmtimes(K,x);
testcond(K);
testsvd(K);
testsolve(K,y);
return
```

C.2.4 Test of **KroneckerProduct2D** and **KroneckerProduct3D**

The tests of the two specialized versions of the Kronecker product, KroneckerProduct2D and KroneckerProduct3D, are similar to the test of the general Kronecker product. We simply use two and three terms, respectively, instead of four. The test of KroneckerProduct2D:

```
function testkron2d
% Test without scaling and transposing
disp('Check 1 - square terms')
AA = randn(8); BB = randn(8); xx = randn(8); yy = randn(8);
A = Matrix(AA); B = Matrix(BB); x = Vector2D(xx); y = Vector2D(yy);
AB1 = KroneckerProduct2D(A,B);

test1(AB1,x,y)

disp('Check 2 - overdetermined ')
AA = randn(8,4); BB = randn(8,4);
xx = randn(4); yy = randn(8);
A = Matrix(AA); B = Matrix(BB); x = Vector2D(xx); y = Vector2D(yy);
AB2 = KroneckerProduct2D(A,B);

test1(AB2,x,y)
```

```
disp('Check 3 - mixed (problems in solve)')
AA = randn(8,4); BB = randn(3,7);
xx = randn(7,4); yy = randn(3,8);
A = Matrix(AA); B = Matrix(BB); x = Vector2D(xx); y = Vector2D(yy);
AB3 = KroneckerProduct2D(A,B);

test2(AB3,x,y)

testopermult(AB1,AB2,AB1);

% ---
function test1(K,x,y)
testmtimes(K,x);
testsvd(K);
testqr(K,0);
testsolve(K,y);
return

% ---
function test2(K,x,y)
testmtimes(K,x);
testsvd(K);
testsolve(K,y);
return
```

Test of KroneckerProduct3D:

```
function testkron3d()

disp('Check 1 - square terms')
AA = randn(4); BB = randn(4); CC = randn(4);
xx = randn(4,4,4); yy = randn(4,4,4);
A = Matrix(AA); B = Matrix(BB); C = Matrix(CC);
x = Vector3D(xx); y = Vector3D(yy);

ABC1 = KroneckerProduct3D(A,B,C);
AABBCC1 = kron(AA,kron(BB,CC));
test1(ABC1,x,y,AABBCC1)

disp('Check 2 - all overdetermined ')
AA = randn(4,3); BB = randn(4,3); CC = randn(4,3);
xx = randn(3,3,3); yy = randn(4,4,4);
A = Matrix(AA); B = Matrix(BB); C = Matrix(CC);
x = Vector3D(xx); y = Vector3D(yy);

AABBCC2 = kron(kron(AA,BB),CC);
ABC2 = KroneckerProduct3D(A,B,C);
test1(ABC2,x,y,AABBCC2)

disp('Check 3 - mixed (problems in solve)')
AA = randn(4,4); BB = randn(3,7); CC = randn(4);
xx = randn(4,7,4); yy = randn(4,3,4);
A = Matrix(AA); B = Matrix(BB); C = Matrix(CC);
x = Vector3D(xx); y = Vector3D(yy);

AABBCC3 = kron(AA,kron(BB,CC));
```

```
ABC3 = KroneckerProduct3D(A,B,C);
test2(ABC3,x,y,AABBCC3)

disp('Test 4 - multiplication');
testopermult(ABC1,ABC2,ABC1);

% ---
function test1(K,x,y,KK)
testgetmatrix(K,KK);
testmtimes(K,x);
testsvd(K);
testqr(K,0);
testsolve(K,y);
return

% ---
function test2(K,x,y,KK)
testgetmatrix(K,KK);
testmtimes(K,x);
testsvd(K);
testsolve(K,y);
return
```

C.2.5 Test of Matrix

The Matrix class encapsulates the ordinary double array and it is assumed that its dimensions are "reasonable" so that the computation of a decompositions such as the SVD can be done. Thus most operations are implemented and must be tested:

```
function testmatrix()

disp('Test 1');
AA = randn(64); A = Matrix(AA);
x = Vector(randn(64,1));

test1(A   ,x,x,  AA);
test1(2*A ,x,x,2*AA);
test1(3*A',x,x,3*AA');
testopermult(A,A,A);
testopermult(2*A',3*A',A);

disp('Test 2');
AA = randn(64,32); A = Matrix(AA);
x1 = Vector(rand(32,1)); x2 = Vector(randn(64,1));

test1(A   ,x1,x2,  AA);
test1(2*A ,x1,x2,2*AA);
test1(3*A',x2,x1,3*AA');
testopermult(A',A,A);
testopermult(2*A',A,A);

disp('Test 3');
AA = randn(32,64); A = Matrix(AA);
x1 = Vector(randn(64,1)); x2 = Vector(randn(32,1));
```

```
test1(A,x1,x2,AA);
test1(2*A,x1,x2,2*AA);
test1(3*A',x2,x1,3*AA');
testopermult(A,2*A',A);

% ---
function test1(A,x,y,AA)
testgetmatrix(A,AA);
testmtimes(A,x);
testsvd(A);
testqr(A);
testcond(A);
testsolve(A,y);
testsize(A);
testdiag(A);
testplusminus(A,A,OperatorSum);
```

Note that we have not implemented `Matrix/plus` and `Matrix/minus`; hence we get an OperatorSum object when two Matrix objects are added.

C.2.6 Test of **NullOperator**

The NullOperator class implements fewer methods than, for instance, the Matrix class. The `sub_solve` method returns a zero vector:

```
function testnulloperator()

AA = zeros(32); A = NullOperator(32);
x = Vector(randn(32,1));

test1(A,x,x,AA)
test1(2*A,x,x,2*AA);
test1(3*A',x,x,3*AA');

AA = zeros(64,32);
x = Vector(randn(32,1)); y = Vector(randn(64,1));
A = NullOperator(y,x);

test1(A,x,y,AA)
test1(2*A,x,y,2*AA)
test1(3*A',y,x,3*AA')

AA = zeros(32,64);
x = Vector(randn(64,1)); y = Vector(randn(32,1));
A = NullOperator(y,x);

test1(A,x,y,AA)
test1(2*A,x,y,2*AA)
test1(3*A',y,x,3*AA')

% ---
function test1(A,x,y,AA)
testgetmatrix(A,AA);
testsize(A);
testcond(A);
```

```
testsolve(A,y);
testmtimes(A,x);
```

C.2.7 Test of OperatorArray

We test the OperatorArray class by initializing a 3×3 block operator consisting of Matrix and IdentityOperator objects.

```
function testoperarray()

AA11 = randn(64); AA12 = randn(64,32);
AA22 = eye(32);   AA21 = randn(32,64);

A11 = Matrix(AA11);          A12 = Matrix(AA12);
A22 = IdentityOperator(32);  A21 = Matrix(AA21);

xx1 = randn(64,1); xx2 = randn(32,1);
xx = [xx1;xx2];

x1 = Vector(xx1); x2 = Vector(xx2);
x = [x1;x2];

AA = [AA11 AA12; AA21 AA22];
A  = [A11 A12; A21 A22];

disp('Test 1');
test1(A,x,AA);

AA = [AA11' 2*AA12; AA21 3*AA22'];
A  = [A11'  2*A12 ; A21  3*A22' ];

disp('Test 2');
test1(A,x,AA);

disp('Test 3 - transposed');
test1(A',x,AA');

disp('Test 4 - operator multiplication');
testopermult(A,A,A);

% ---
function test1(A,x,AA)
testgetmatrix(A,AA)
testsize(A);
testmtimes(A,x);
testplusminus(A,A,OperatorSum);
```

C.2.8 Test of OperatorProduct

The OperatorProduct class implements `sub_solve` for products where all terms are square (and of equal size). No decompositions are defined:

```
function testoperproduct
```

```
disp('Test 1')
AA = randn(8); BB = randn(8);
A  = Matrix(AA); B = Matrix(BB);
xx = randn(8,1); x = Vector(xx);
yy = randn(8,1); y = Vector(yy);
AB = OperatorProduct({A,B});

test1(AB,x,y,AA*BB);
test1(AB',x,y,BB'*AA');

disp('Test 2 - tranpose')
AB = OperatorProduct({2*A,B'});
test1(AB,x,y,2*AA*BB')

disp('Test 3 - extending product')
ABB = AB*B;
test1(ABB,x,y,2*AA*BB'*BB)

disp('Test 4 - non-square terms')
AA = randn(8,10); BB = randn(10,8);
A  = Matrix(AA); B = Matrix(BB);
xx = randn(8,1); x = Vector(xx);
AB = OperatorProduct({A,B});
test2(AB,x,AA*BB);

disp('Test 5 - extracting columns')
testextractcolumns(AB,1:4)

% ---
function test1(A,x,y,AA)
testgetmatrix(A,AA)
testsize(A);
testmtimes(A,x)
testsolve(A,y);
testplusminus(A,A, OperatorSum);
return

% ---
function test2(A,x,AA)
testgetmatrix(A,AA)
testsize(A);
testmtimes(A,x)
testplusminus(A,A, OperatorSum);
```

C.2.9 Test of **OperatorSum**

The OperatorSum class only implements multiplication with a vector:

```
function testopersum()

AA = randn(8); BB = randn(8); xx = rand(8,1);
A = Matrix(AA); B = Matrix(BB); x = Vector(xx);

disp('Test 1');
AABB = AA+BB; AB = OperatorSum({A,B});
```

```
test1(AB,x,AABB);

disp('Test 2');
AABB = 2*AA+3*BB'; AB = OperatorSum({2*A, 3*B'});
test1(AB,x,AABB);

disp('Test 3 - transposed');
test1(AB',x,AABB');

function test1(A,x,AA)
testgetmatrix(A,AA);
testsize(A);
testmtimes(A,x);
```

C.2.10 Test of OperatorSVD

The OperatorSVD class is used to cache the SVD factors. We test if the proper terms are returned and if the multiplication operation with the entire object is done properly:

```
function testoperatorsvd()

AA = randn(64); A = Matrix(AA);
[U,S,V] = svd(A); [UU,SS,VV] = svd(AA);
Asvd = OperatorSVD(U,S,V);
x = Vector(randn(64,1));

test1(Asvd,x,AA);
test1(2*Asvd', x , 2*AA');

test1(getU(Asvd),x,UU);
test1(getV(Asvd),x,VV);
test1(getS(Asvd),x,SS);

% ---
function test1(U,x,UU)

testgetmatrix(U,UU);
testmtimes(U,x);
testsolve(U,x)
testsvd(U);
```

C.2.11 Test of PermuationOperator

The PermutationOperator shuffles the elements of Vector objects. We perform the usual set of tests:

```
function testpermutation()

v = rand(100,1);
[vv,idx] = sort(v);
v = Vector(v);

PP = eye(100); PP = PP(:,idx);
```

```
P = PermutationOperator(idx);

disp('Test 1')
test1(P,v,v,PP)
test1(2*P,v,v,2*PP)
test1(2*P',v,v,2*PP')

disp('Test 2 - extract columns')
P = P(:,1:30); PP = PP(:,1:30); v2 = v(1:30);
test1(P   ,v2,v ,  PP)
test1(2*P ,v2,v ,2*PP)
test1(2*P',v ,v2,2*PP')

% ---
function test1(A,x,y,AA)
testgetmatrix(A,AA);
testsize(A);
testmtimes(A,x);
testmtimes(A',y);
```

C.2.12 Test of SparseMatrix

The SparseMatrix class implements fewer methods than the dense Matrix class. For instance we do not have decompositions such as the SVD:

```
function testsparse()

AA = sprand(64,64,0.4,1e-4); A = SparseMatrix(AA);
x = Vector(randn(64,1));

disp('Test 1')
test1(A   ,x,x,  AA);
test1(2*A ,x,x,2*AA);
test1(3*A',x,x,3*AA');

disp('Test 2');
AA = sprandn(64,32,0.2); A = SparseMatrix(AA);
x1 = Vector(rand(32,1)); x2 = Vector(randn(64,1));

test1(A   ,x1,x2,  AA);
test1(2*A ,x1,x2,2*AA);
test1(3*A',x2,x1,3*AA');

disp('Test 3');
AA = sprandn(32,64,0.3); A = SparseMatrix(AA);
x1 = Vector(randn(64,1)); x2 = Vector(randn(32,1));

test1(A,x1,x2,AA);
test1(2*A,x1,x2,2*AA);
test1(3*A',x2,x1,3*AA');

% ---
function test1(A,x,y,AA)
testgetmatrix(A,AA);
```

```
testmtimes(A,x);
testsolve(A,y);
testsize(A);
testdiag(A);
testplusminus(A,A,OperatorSum);
```

C.2.13 Test of TikhPrecond

The preconditioning class TikhPrecond only implements `sub_solve`. Results are compared to results obtained from an explicitly created double array:

```
function testtikhprecond()
AA = randn(64); A = Matrix(AA); l = 1e-3;

[UU,SS,VV] = svd(AA); ss = diag(SS);
PP = VV*diag( sqrt(ss.^2 + l^2) )*VV';

P = TikhPrecond(A, l); x = Vector(randn(64,1));

test1(P  ,x,PP);
test1(P' ,x,PP');
test1(2*P,x,2*PP);

%-----
function test1(A,x,AA)
testgetmatrix(A,AA);
testsize(A);
testsolve(A,x);
```

C.2.14 Test of VectorCollection

We create a VectorCollection from the columns of a double array and check the usual set of multiplication operations:

```
function testvectorcollection()

AA = randn(64); xx = randn(64,1); x = Vector(xx);
A = VectorCollection(64);
for i=1:64,  A(:,i) = Vector(AA(:,i)); end

disp('Test 1');
test1(A,x,AA);

disp('Test 2');
test1(A',x,AA');

disp('Test 3');
test1(2*A',x,2*AA')

disp('Test 4 - extract cols');
x = Vector(randn(32,1)); y = Vector(randn(64,1));
A = A(:,1:32); AA = AA(:,1:32);
test1(A, x, AA);
test1(2*A', y, 2*AA');
```

```
disp('Tet 5 - operator mult')
KK = randn(64); K = Matrix(KK);
test1(K*A,x,KK*AA);

% ---
function test1(A,x,AA)
testgetmatrix(A,AA);
testsize(A);
testmtimes(A,x);
```

C.2.15 Test of VectorReshape

To test VectorReshape we see if it changes the type of vectors as intended:

```
function testvectorreshape()

disp('Test 1 - Vector - Vector2D')
xx = randn(64); X1 = Vector(xx(:)); X2 = Vector2D(xx);
R = VectorReshape(64,64);
test1(R,X1,X2);
test1(R',X2,X1);

disp('Test 2 - Vector - Vector3D')
xx = randn(64,64,32); X1 = Vector(xx(:)); X2 = Vector3D(xx);
R = VectorReshape(64,64,32);
test1(R,X1,X2);
test1(R',X2,X1);

disp('Test 3 - Vector - VectorND')
xx = randn(64,64,64,32); X1 = Vector(xx(:)); X2 = VectorND(xx);
R = VectorReshape(64,64,64,32);
test1(R,X1,X2);
test1(R',X2,X1);

% ---
function test1(R,X1,X2)
if class(R*X1) ~= class(X2), printerror('type', R); end
if norm(R*X1 - X2) ~= 0,    printerror('mtimes', R);  end
```

C.2.16 Test of WeightedPseudoInverse

We check the WeightedPseudoInverse class against a double array created explicitly from the GSVD:

```
function testweightedpseudoinverse()

KK = randn(64,32); K = Matrix(KK);
[L,W] = getL(32,1); LL = getmatrix(L); WW = getmatrix(W);

LKinv = WeightedPseudoInverse(L,W,K);

[UU,sm,XX,VV] = cgsvd(KK,LL);
MM = full(spdiags(sm(:,2),0,size(L,1),size(L,2)));
```

```
LLKKinv = XX*pinv(MM)*VV';

x = Vector(randn(31,1));
y = Vector(randn(32,1));
test1(LKinv, x, LLKKinv);
test1(LKinv', y, LLKKinv');

d = norm(nullcomp(LKinv, K*y) - pinv(KK*WW*WW')*getvector(K*y));
if d > eps*32
  printerror('nullcomp', LKinv, d);
end

% ---
function test1(K,x,KK,WW)
testgetmatrix(K,KK);
testsize(K);
testmtimes(K,x);
```

C.3 Testing the Algorithms

The tests of the algorithms are listed in alphabetical order which, in some cases, means that we refer to tests of functions/algorithms described later in this appendix. The algorithms that combine modules are only tested for syntax errors as they are based on the tests of the individual modules. They have also been subject to a code review.

Test of `ahybrid`

We test if the Arnoldi/upper Hessenberg decomposition within the hybrid method has the correct properties and if the regularization of the inner problem has been carried out correctly:

```
function testahybrid()

[K,y,x] = shaw(32);

inopt = regset('RegPar',1e-2);
opt = regset('Iter',10,'InSolv','tikhonov','InArgs',inopt,'Subspc','on');
[X, extra] = ahybrid(K,y,opt);

VV = getmatrix(extra.V);
HH = getmatrix(extra.H);
KK = getmatrix(K);

t = KK*VV(:,1:end-1)-VV*HH;

d = norm(t,'fro');
if d > eps*100, printerror('ahybrid 1', K, d); end

d = norm(HH - triu(HH,-1), 'fro');
if d > 0, printerror('ahybrid 2',K); end
```

```
X1 = extra.V(:,1:10)*tikhonov(extra.H, extra.V'*y, inopt);
d = norm(X-X1);
if d > length(X)*eps, printerror('ahybrid 4',K,d), end
```

Test of `cg`

The test of conjugate gradients compares its result with `lsqr_mt` (see later) after a few iterations, such that rounding errors are limited. We use a symmetric positive definite test problem obtained from the normal equations.

```
function testcg()

[K,y,x] = deriv2(32);

A = K'*K; b = K'*y;

[X1, ext1] = cg(A,b,regset('Iter',4,'TolRes',eps));
[X2, ext2] = lsqr_mt(K,y,regset('Iter', 4, 'TolRes', eps));

d = norm(X1-X2);
if d > 1e-10, printerror('cg', K, d); end
```

Test of `cgls`

The `cgls` function is tested against the `lsqr_mt` function (tested later).

```
function testcgls()
[K,y,x] = deriv2(32);
[X1, ext1] = cgls(K,y,regset('Iter', 4,'TolRes', eps, 'Reorth', 'on'));
[X2, ext2] = lsqr_mt(K,y,regset('Iter', 4,'TolRes', eps, 'Reorth', 'on'));

d = norm(X1-X2);
if d > 1e-10, printerror('cgls', K, d); end

[X1,ext1,restart] = cgls(K,y,regset('Iter', 2));
[X1,ext1,restart] = cgls(K,y,regset('Iter', 2),restart);

d = norm(X1-X2);
if d > 1e-10, printerror('cgls restart', K, d); end
```

Test of `gmin`

We test `gmin` by comparing its result with the result from `fminunc` from the Optimization Toolbox. We use `backslash` (Gauss-elimination as provided by Matlab) to solve the inner problems. Hereby we try to only check `gmin` and not the accuracy and other problems, such as the choice of stopping criteria, associated with an iterative inner solver. However, we do not expect the results to be exactly the same due to the different methods used:

```
function testgmin()
[K,y,x] = deriv2(32);
```

```
y = y + randn(size(y))*1e-3; y(4) = y(4)*3;

KL = [K; IdentityOperator(size(K,2))]; y0  = [y; zeros(y)];

normfun = 'fair'; normarg = 1e-3;
options = regset('InSolv', 'backslash', ...
                 'InArgs', [], ...
                 'Iter', 50, ...
                 'Norm', normfun, ...
                 'NormArgs',normarg);

[x1,extra] = gmin(KL,y0,options);

opt = optimset('GradObj', 'on','TolFun',1e-10,'MaxIter',50,...
               'Hessian','on','TolX',eps);
xx1 = fminunc(@fun, zeros(size(x1)), opt,KL,y0,normfun,normarg);

d = norm(xx1-x1)/norm(x1);

if d > 1e-6, printerror('gmin solution', K, d); end


normfun = 'stacknorm'; normarg =  {'fair', 1e-1, 1; 'logistic', 1e-1, 1e-3};
options = regset(options, ...
                 'Norm', normfun, ...
                 'NormArgs', normarg);

[x2, extra] = gmin(KL,y0,options);
xx2 = fminunc(@fun, zeros(size(x1)), opt,KL,y0,normfun,normarg);

d = norm(xx2-x2)/norm(xx2);
if d > 1e-6, printerror('gmin solution', K, d); end

% ---
function [x,dx,ddx] = fun(x, KL, b0,normfun, normarg,xx)

[n,dn,ddx] = feval(normfun, KL*Vector(x)-b0, normarg);

x = n;
dx = getvector(KL'*dn);
ddx = getmatrix(KL)'*diag(getvector(ddx))*getmatrix(KL);
```

Test of gminnonneg

The non-negativity constrained minimization performed by gminnonneg is compared to the result of fmincon from the Optimization Toolbox. Also in this case we expect a larger discrepancy due to the different methods used:

```
function testgminnonneg()
n=100;

[K,y,x] = heat(n);
y = y + randn(size(y))*1e-3; y(4) = y(4)*3;

KL = stack(K,1e-3*IdentityOperator(n)); y0 = stack(y,zeros(y));
```

```
normfun = 'fair'; normarg = 0.001;
inargs = regset('Iter', 55, 'TolRes', 1e-12,'Disp',0);
options = regset('Iter', 20, ...
                 'InSolv', 'lsqr_mt', ...
                 'InArgs', inargs, ...
                 'Norm', normfun, ...
                 'NormArgs', normarg);

[x1, extra] = gminnonneg(KL,y0,options);

opt = optimset('GradObj', 'on', 'TolFun', 1e-10, 'MaxIter', 50, ...
               'Hessian', 'on', 'TolX', eps);
xx1 = fmincon(@fun, zeros(size(x1)), ...
              [],[],[],[],zeros(size(x1)),[],[], ...
              opt,KL,y0,normfun,normarg);

n = feval(normfun, KL*x1 - y0, normarg);
nn = feval(normfun, Vector(getmatrix(KL)*xx1-getvector(y0)), normarg);

if (n-nn)/nn > 1e-2, printerror('gminnonneg solution', K, (n-nn)/nn); end

% ---
function [x,dx,ddx] = fun(x, KL, b0,normfun, normarg)

[n,dn,ddx] = feval(normfun, KL*Vector(x)-b0, normarg);

x = n;
dx = getvector(KL'*dn);
ddx = getmatrix(KL)'*diag(getvector(ddx))*getmatrix(KL);
```

Test of `gmres_mt`

The GMRES function is tested against the result from the Matlab implementation of GMRES after 4 iterations:

```
function testgmres()

[A,b,x] = deriv2(32);

[X,extra] = gmres_mt(A,b,regset('Iter',4,'Subspc', 'on'));
[xx,flag] = gmres(getmatrix(A),getvector(b),4,eps,1);

d = norm(X-xx);
if d > length(xx)*eps, printerror('gmres solution', A, d); end

VV = getmatrix(extra.V);
d = norm( VV*((getmatrix(A)*VV)\getvector(b)) - getvector(X));
if d > length(xx)*eps, printerror('gmres subs',K,d); end
```

Test of `l1sol`

The least absolute sum minimization is tested by comparing the result to the result of `l1c` (see the introduction to this appendix). Because `l1sol` is based on the interior point method in `linprog` of the Optimization toolbox and `l1c` uses the simplex algorithm we expect a small difference in the results:

```
function testl1sol()

KK = randn(10,5); yy = randn(10,1);
K = Matrix(KK); y = Vector(yy);

x = l1sol(K,y,[]);
xx = l1c(KK,yy,[],[],[],[]);

d = norm(K*x-y,1) - norm(KK*xx-yy,1);
if d < 0, printerror('l1sol solution', K, d); end
```

Test of `l1tsvd`

The test of `l1tsvd` is similar to the test for `l1sol`:

```
function testl1tsvd()

[K,y,x] = deriv2(32); KK = getmatrix(K); yy = getvector(y);
[U,S,V] = svd(K);

x = l1tsvd(K,y,regset('RegPar',10));

UU = getmatrix(U); SS = getmatrix(S); VV = getmatrix(V);
VV10 = VV(:,1:10);

xx = VV10*l1c(KK*VV10,yy,[],[],[],[]);

d = norm(VV10*(VV10'*getvector(x))-getvector(x));
if d > length(x)*eps, printerror('l1tsvd subspc',K,d); end

d = norm(K*x-y,1) - norm(KK*xx-yy,1);
if d > 0, printerror('l1tsvd solution',K,d); end
```

Test of `lptsvd`

We test the `lptsvd` function with a 1.5-norm problem. We check the correctness by looking at the gradient at the result:

```
function testlptsvd()

p = 1.5;

[K,y,x] = deriv2(32); KK = getmatrix(K); yy = getvector(y);
[U,S,V] = svd(K);

opt = regset('RegPar',10, 'Norm', 'pnorm', 'NormArgs', p, ...
             'TolRes',1e-8, 'Iter', 20);
x = lptsvd(K,y,opt);

VV = getmatrix(V); VV10 = VV(:,1:10);
```

```
d = norm(VV10*(VV10'*getvector(x))-getvector(x));
if d > length(x)*eps, printerror('lptsvd subspc',K,d); end

r = KK*getvector(x)-yy;
ng = norm(VV10'*KK'*(p*abs(r).^(p-1).*sign(r)));
if ng > 1e-8, printerror('l1tsvd solution',K,ng); end
```

Test of `lbhybrid`

The hybrid method based on Lanczos bidiagonalization is compared with LSQR (using no inner regularization) and LSQR on a damped least squares problem (corresponding to Tikhonov regularization on the inner problem):

```
function testlbhybrid()

[K,y,x] = shaw(32);

inopt = regset('RegPar',0);
opt = regset('Iter',4,'InSolv','tikhonov','InArgs',inopt,'Reorth','on');
X  = lbhybrid(K,y,opt);
X1 = lsqr_mt(K,y,regset('Iter', 4, 'Reorth','on'));

d = norm(X1-X);
if d > eps*100, printerror('lbhybrid 1', K, d); end

inopt = regset('RegPar', 1e-2);
opt = regset('Iter',5,'InSolv','tikhonov','InArgs',inopt,'Reorth','on');
[X, extra] = lbhybrid(K,y,opt);

KL = stack(K, 1e-2*IdentityOperator(32)); y0 = stack(y, zeros(y));
X1 = lsqr_mt(KL,y0,regset('Iter', 5, 'Reorth','on'));

d = norm(X1-X);
if d > eps*100, printerror('lbhybrid 2', K, d); end
```

Test of `lsqr_mt`

We compare the result of `lsqr_mt` with `lsqr_b` from the Regularization Tools package. Then we test the restart feature and the preconditioning feature:

```
function testlsqr()

[K,y,x] = deriv2(64);
iter = 5;

opt = regset('Iter', iter, 'TolRes',eps, 'Reorth', 'on', 'Subspc','on');
[X1, ext1] = lsqr_mt(K,y,opt);
X = lsqr_b(getmatrix(K),getvector(y),iter,1);

d = norm(getvector(X1)-X(:,end));
if d > eps*cond(K), printerror('lsqr solution', K, d); end

VV = getmatrix(ext1.V);
d = norm( VV*((getmatrix(K)*VV)\getvector(y)) - getvector(X1));
```

```
if d > eps*64, printerror('lsqr subspace',K,d); end

% Test restart do 2*5 and 10 iterations
opt = regset('Iter', 5);
[X1, ext1,restart] = lsqr_mt(K,y,opt);
[X2, ext2] = lsqr_mt(K,y,opt,restart);
opt = regset('Iter', 10);
[X3, ext3] = lsqr_mt(K,y,opt);

d = norm(X2 - X3);
if d > eps*cond(K), printerror('lsqr restart', K, d); end

P = Matrix(2*diag([ones(32,1);2*ones(32,1)]));
opt = regset('Iter',iter,'Precond',P, 'Reorth', 'on','TolRes',eps);
X1 = lsqr_mt(K,y,opt);
X = lsqr_b(getmatrix(K*inv(P)),getvector(y),iter,1);

d = norm(X1 - inv(P)*X(:,end));
if d > eps*cond(K), printerror('lsqr precond',K,d); end
```

Test of `lthybrid`

The hybrid method based on Lanczos tridiagonalization is equivalent with the MINRES method if we do not regularize the inner problem.

```
function testlthybrid()

[K,y] = deriv2(32); K = K; y = y;
KK = getmatrix(K); yy = getvector(y);

inopt = regset('RegPar',0);
opt = regset('Iter',2,'Subspc','on','Reorth','on', ...
             'InSolv','tikhonov','InArgs',inopt);
[X1, ext1] = lthybrid(K,y,opt);
[X2, ext2] = minres_mt(K,y,opt);

d = norm(X1-X2);
if d > eps*100, printerror('lthybrid 1', K, d); end

inopt = regset('RegPar', 1e-2);
opt = regset('Iter',5,'InSolv','tikhonov','InArgs',inopt,'Reorth','on',...
             'Subspc','on');
[X1, ext1] = lthybrid(K,y,opt);
X2 = ext1.V*tikhonov(K*ext1.V,y,regset('RegPar',1e-2) );

d = norm(X1-X2);
if d > eps*100, printerror('lthybrid 2', K, d); end
```

Test of `minres_mt`

The MINRES method `minres_mt` is compared to results from Matlab's implementation of MINRES:

```
function testminres()
```

```
AA = orth(randn(32)); D = diag(randn(32,1));
AA = AA'*D*AA; xx = randn(32,1);

A = Matrix(AA); x = Vector(xx); b = A*x;

[x,ext] = minres_mt(A,b,regset('Iter',10));
[xx,ext] = minres(AA,getvector(b),eps,9); % Matlab count iterations differently

d = norm(x-xx);
if d > length(x)*eps, printerror('minres', A, d); end
```

Test of `mrII`

The `mrII` method is a variation of MINRES. We check if the solution lies in the correct subspace obtained from `lthybrid`:

```
function testmrii()

AA = orth(randn(32)); D = diag(randn(32,1));
AA = AA'*D*AA; xx = randn(32,1);

A = Matrix(AA); x = Vector(xx); b = A*x;

[x,ext1] = mrII(A,b,regset('Iter',10, 'Subspc', 'on'));

inopt = regset('RegPar', 0);
opt = regset('InSolv','tikhonov','Subspc','on','InArgs', inopt,'Iter',10,...
             'v1', A*b, 'Reorth','on');
[xx,ext2] = lthybrid(A,b,opt);

VV1 = orth(getmatrix(ext1.V)); VV2 = orth(getmatrix(ext2.V));

d = norm(VV1'*VV1-VV2'*VV2);
if d > length(x)*eps, printerror('mrII subspace', A, d); end

xx3 = VV1*( (getmatrix(A)*VV1)\getvector(b) );

d = norm(x-xx3);
if d > 100*eps, printerror('mrII solution'); end
```

Test of `tikhonov` (LinearOperator)

We force M$\mathcal{OO}$Re Tools to use the LinearOperator implementation of Tikhonov regularization by converting a problem into a SparseMatrix. A SparseMatrix does not support the SVD and thus it uses the LSQR-based Tikhonov algorithm found in LinearOperator:

```
function testtikhonovlo()

[K1,y] = deriv2(32);

K2 = SparseMatrix(sparse(getmatrix(K1)));

opt = regset('RegPar', 1e-2);
```

```
x = tikhonov(K2,y,opt);
xx = tikhonov(K1,y,opt);

d = norm(x-xx);
if d > eps*100, printerror('tikhonov(LO)',K2,d); end
```

Test of `tikhonov` (**SVDOperator**)

The results of the Tikhonov solver based on the SVD (and thus a method of SVDOperator is compared to the results of the Tikhonov solver from the previous toolbox:

```
function testtikhonovsvd()

[K,y] = deriv2(64);
yn = y + randn(size(y))*1e-3;

x = tikhonov(K,y, regset('RegPar',1e-3));
[U,s,V] = csvd(getmatrix(K));
xx = tikhonov(U,s,V,getvector(y),1e-3);

d = norm(x-xx);
if d > eps*64, printerror('tikhonov(svd)',K,d); end

[x,ext] = tikhonov(K,yn,regset('RegPar','gcv'));
xx = tikhonov(U,s,V,getvector(yn),ext.RegPar);

d = norm(x-xx);
if d > eps*64, printerror('tikhonov(svd) gcv',K,d); end
```

Test of `tsvd`

We compare the TSVD method with the corresponding function in the previous toolbox:

```
function testtsvd()

[K,y,x] = deriv2(64);
y = y + randn(size(y))*1e-3;
ks = 10:20;

opt = regset('RegPar', ks);

X = tsvd(K,y,opt);

[UU,SS,VV] = csvd(getmatrix(K));
XX = tsvd(UU,SS,VV,getvector(y), ks);

d = norm(XX-getmatrix(X), 'fro');
if d > 100*eps,  printerror('tsvd solutions',K,d); end

[x, ext] = tsvd(K,y,regset('RegPar', 'gcv'));
xx = tsvd(UU,SS,VV,getvector(y), ext.RegPar);
if d > 100*eps, printerror('tsvd par choice', K, d); end
```

C.3.1 Test of Parameter Choice Methods

Test of gcv

The GCV method is compared with the GCV function from the previous toolbox (from which it is derived):

```
function testgcv()

[K,y] = deriv2(32);
yn = y + randn(size(y))*1e-3;

[UU,ss,VV] = csvd(getmatrix(K));

reg1 = gcv(K,yn,'Tikh');
reg2 = gcv(UU,ss,getvector(yn),'Tikh');

if reg1 ~= reg2,
  printerror('gcv error',K,abs(reg1-reg2)/max(reg1,reg2))
end
```

Test of gcvbounds

Because the gcvbounds method computes bounds for an approximation to the trace-term of the GCV-function we compute the exact quantity obtained with the MC technique to check if the bounds are correct:

```
function testgcvbounds()

[K,y] = deriv2(64);
yn = y + randn(size(y))*1e-3*norm(y);


[reg, extra] = gcvbounds(K,yn, regset('RegPar', [1e-6 1], 'TolRes', 1e-3));
l = extra.lambda;

KK = getmatrix(K); yy = getvector(yn);
[UU,ss,VV] = csvd(KK);
[x,rho,eta] = tikhonov(UU,ss,VV,yy,l);

I = eye(64);
uu = getvector(extra.u);
traceapprox = zeros(length(l),1);
for i=1:length(l)
  traceapprox(i) = l(i)^2*uu'*inv(KK*KK'+l(i)^2*I)*uu;
end

d = min(rho - extra.rhoL);
if d < -100*eps,  printerror('rho lower',K,d); end

d = min(extra.rhoU - rho);
if d < -100*eps,  printerror('rho upper',K,d); end

d = min(traceapprox - extra.traceL);
if d < -100*eps,  printerror('trace lower',K,d); end
```

```
d = min(extra.traceU - traceapprox);
if d < -100*eps,  printerror('trace upper',K,d); end

d = min(rho./traceapprox - extra.gcvL);
if d < -100*eps, printerror('gcv lower',K,d); end

d = min(extra.gcvU - rho./traceapprox);
if d < -100*eps, printerror('gcv lower',K,d); end
```

Test of `l_curve`

The L-curve method is compared with the function from the previous toolbox (from which it is adapted):

```
function testlcurve()

[K,y] = deriv2(32);
yn = y + randn(size(y))*1e-3;
KK = getmatrix(K); yy = getvector(yn);
[U,s,V] = csvd(KK);

regpar1 = l_curve(K,yn,'Tikh');
regpar2 = l_curve(U,s,yy,'Tikh');

d = regpar1-regpar2;
if d ~= 0, printerror('lcurve', K, d); end
```

Test of `lcurvebounds`

We check the L-curve curvature bounds against the curvature computed by the auxiliary function `lcfun` found in the previous toolbox:

```
function testlcurvebounds()

[K,y,x] = deriv2(64); yn = y + randn(size(y))*1e-3;

opt = regset('Iter', 40, 'RegPar', [1e-4 1], 'TolRes', 1e-1, 'Reorth','on');
[reg, ext] = lcurvebounds(K, yn , opt);
l = ext.lambda;

KK = getmatrix(K); yy = getvector(yn);

[UU,ss,VV] = csvd(KK);
[X,rho,eta] = tikhonov(UU,ss,VV,yy,l);

beta = UU'*yy; xi = beta./ss;
cur = lcfun(l,ss,beta,xi);

d = min(rho - ext.rhoL);
if d < -100*eps, printerror('lc bounds rhoL', K,d); end

d = min(eta - ext.etaL);
if d < -100*eps, printerror('lc bounds etaL', K,d); end
```

```
d = min(cur - ext.curL);
if d < -100*eps, printerror('lc bounds curL', K,d); end

d = min(ext.rhoU - rho);
if d < -100*eps, printerror('lc bounds rhoU', K,d); end

d = min(ext.etaU - eta);
if d < -100*eps, printerror('lc bounds etaU', K,d); end

d = min(ext.curU - cur);
if d < -100*eps, printerror('lc bounds cirU', K,d); end
```

Test of `reginska`

The Regińska parameter-choice method is tested using exact values of solution and residual norm obtained from a Tikhonov computation. We see if we get the upper and lower bounds we expect:

```
function testreginska()

[K,y] = deriv2(32);
yn = y + randn(size(y))*1e-3;

[regpar, ext] = reginska(K,yn,regset('RegPar',[1e-5 1],'TolRes',1e-3));
[X, extt] = tikhonov(K,yn, regset('RegPar', ext.lambda));

phi = extt.rho .* extt.eta;
d = min(min(phi - ext.phiL),min(ext.phiU - phi));
if d > 100*eps,  printerror('reginska', K);end
```

C.3.2 Test of Support Functions

Of the supporting methods listed in Table B.3 we do not test `regset` and `regget` as they have been used extensively in almost all tests in this appendix. The function `linesearch` is not tested because we have copied it from Nielsen's homepage `www.imm.dtu.dk/~hbn/Software` (the algorithm is described in [40]) and is assumed correct. The assumption is supported by the fact that the methods that use `linesearch`, such as `gmin`, return reasonable results.

Test of `gmodsubs`

We test minimization in general norm with orthogonality constraints against `fmincon` from the Optimization Toolbox:

```
function testgmodsubs()

[K,y,x] = deriv2(100);
[x0,extra] = lsqr_mt(K,y,regset('Iter',5,'Subspc','on', 'Reorth','on'));

L = getL(100,1);
```

```
inargs = regset('TolRes', 1e-10);
normfun = 'fair'; normargs = 0.5;
opt = regset('Norm', normfun, 'NormArgs', normargs, ...
 'Iter',10, 'InArgs', inargs);
xp = gmodsubs(L, extra.V, x0, opt);
d = norm(extra.V'*xp);
if d > length(xp)*eps, printerror('gmodsubs ortho', K, d); end

LL = full(getmatrix(L)); VV = getmatrix(extra.V); xx0 = getvector(x0);

opt = optimset('GradObj','on');
xxp = fmincon(@fun, zeros(size(xp)), [],[], VV', zeros(size(VV,2),1), ...
[],[],[],opt,LL,-LL*xx0,normfun, normargs);

d = norm(xp - xxp);
if d > length(xp)*eps, printerror('gmodsubs sol', K, d); end

n1 = feval(normfun, L*(xp+x0), normargs);
n2 = feval(normfun, L*(xxp+x0), normargs);
d  = n2 - n1;
if d < 0, printerror('gmodsubs sol 2', K, d); end

% ---
function [n,ndx] = fun(x, K, y,normfun,normargs)

r = K*x-y;
[n,dn] = feval(normfun, Vector(r), normargs);
ndx = K'*getvector(dn);
```

Test of `l1orth`

The result from `l1orth` is compared to the result from `l1c` (see introduction to this appendix):

```
function testl1orth()

AA = randn(32); VV = randn(32,10); bb = randn(32,1);
A = Matrix(AA); V = Matrix(VV); b = Vector(bb);

x = l1orth(A,b,V',zeros(V'*b));
xx = l1c(AA,bb,VV',zeros(10,1),[],[]);

d = norm(x - xx);
if d > eps*length(x), printerror('l1orth',A,d); end
```

Test of `lsorth`

The constrained least squares solver `lsorth` is compared to `lsqlin` from the Optimization Toolbox:

```
function testlsorth()

AA = randn(32); VV = randn(32,10); bb = randn(32,1);
A = Matrix(AA); V = Matrix(VV); b = Vector(bb);
```

```
x = lsorth(A,b,V',zeros(V'*b),regset('TolRes',eps));
xx = lsqlin(AA,bb,[],[],VV',zeros(size(V,2),1));

d = norm(x - xx);
if d > eps*length(x), printerror('lsorth',A,d); end
```

Test of modsubs

The function modsubs uses lsorth and also here we use lsqlin to check the result.

```
function testmodsubs()

[K,y,x] = deriv2(100);
[x0,extra] = lsqr_mt(K,y,regset('Iter',5,'Subspc','on', 'Reorth','on'));

L = getL(100,1);

xp = modsubs(L, extra.V, x0, regset('TolRes', eps,'Iter',200));
d = norm(extra.V'*xp);
if d > length(xp)*eps, printerror('modsubs ortho', K, d); end

LL = full(getmatrix(L)); VV = getmatrix(extra.V); xx0 = getvector(x0);
opt = optimset('LargeScale', 'off');
xxp = lsqlin(LL,-LL*xx0, [],[],VV', zeros(size(VV,2),1),[],[],[],opt );

d = norm(xp - xxp);
if d > length(xp)*eps, printerror('modsubs sol', K, d); end
```

Test of modsubs1

The result of modsubs1 is compared to the result of l1c from Regularization Tools:

```
function testmodsubs1()

[K,y,x] = deriv2(100);
[x0,extra] = lsqr_mt(K,y,regset('Iter',5,'Subspc','on', 'Reorth','on'));

L = getL(100,1);

xp = modsubs1(L, extra.V, x0);
d = norm(extra.V'*xp);
if d > length(xp)*eps, printerror('modsubs1 ortho', K, d); end

LL = getmatrix(L); VV = getmatrix(extra.V); xx0 = getvector(x0);
xxp = l1c(LL,-LL*xx0, VV', zeros(size(VV,2),1),[],[]);

d = norm(xp - xxp);
if d > length(xp)*eps, printerror('modsubs1 sol', K, d); end
```

C.3.3 The Combined Algorithms

The algorithms obtained by combining modules are tested for correct syntax and has been under code review, that is, the code has been looked at with extra care and the results have been checked to see if they appear reasonable.

Test of `glsqr`

```
function testglsqr()

[K,y,x] = deriv2(32); L = getL(32,1);

opt = regset('Iter',10, 'Norm','pnorm','NormArgs',1.4);

X = glsqr(K,L,y,opt);
```

Test of `mlsqr`

```
function testmlsqr()

[K,y,x] = deriv2(32);
L = getL(32,1);

x = mlsqr(K,L,y, regset('Iter',10));
```

Test of `mtsvd`

```
function testmtsvd()

[K,y,x] = deriv2(32); L = getL(32,1);
opt = regset('Iter',10);
X = mtsvd(K,L,y,opt);
```

Test of `ppgmres`

```
function testppgmres()

[K,y,x] = deriv2(32); L = getL(32,1);

opt = regset('Iter',10);

X = ppgmres(K,L,y,opt);
```

Test of `pplsqr`

```
function testpplsqr()

[K,y,x] = deriv2(32); L = getL(32,1);
opt = regset('Iter',10);
X = pplsqr(K,L,y,opt);
```

Appendix D

New Test Problems

All test problems from Regularization Tools [67] have been converted to use the objects. In most cases they simply use Matrix and Vector. In the following the new test problems are briefly explained.

D.1 Steady-State Heat Distribution

Consider a square area with a circular cut out, see Fig. D.1. The outer boundary is perfectly insulated. The inner boundary Γ_1 has a fixed temperature distribution described by $g(x)$. The forward problem is to find the temperature on the outer boundary Γ_2 given the temperature distribution at the inner boundary. The inverse problem is to determine the temperature function on the inside from measurements on the outer boundary.

Put into a partial differential framework we have the following setup

$$\begin{cases} \nabla^2 t(x,y) = 0 & (x,y) \in \Omega \\ t(x,y) = g(x,y) & (x,y) \in \Gamma_1 \\ \frac{d}{dn} t(x,y) = 0 & (x,y) \in \Gamma_2. \end{cases} \tag{D.1}$$

Each forward calculation implies solving the partial differential equation (PDE) system. We consider three methods:

Finite differences. Finite differences is perhaps the easiest to understand. The derivatives are approximated with approximations on a simple grid of points in the area. On the other hand the finite difference method is troublesome when

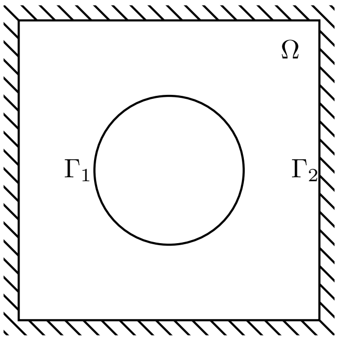


Figure D.1: The geometry of the steady state heat problem.

boundaries become complicated. The method also finds the solution within Ω. See [9] for an example of applying finite differences to a problem of the same type albeit another geometry. We have not implemented the finite difference method.

Finite element method (FEM). The finite element method divides Ω into small sub-areas, where a set of basis functions describes an approximation to the solution. The FEM method is very flexible with respect to the geometry of the problem. The FEMLAB [27] Matlab package uses object oriented techniques to describe the geometry. See also [89], where an object oriented toolbox for partial differential equations using FEM (and the C++ language) is described in the same spirit as this thesis. The SteadyHeatFEM class is implemented using the FEMLAB package and we refer to its documentation for details on the solution method.

Boundary element method (BEM). The boundary element method is different from both the finite difference method and FEM as it only computes unknown values at the boundaries—at least in its first steps. Because the code is not based on a existing package, we will describe the BEM method in more detail for our particular problem.

If we consider the Laplace equation $\nabla^2 t = 0$ in two dimensions and introduce a weighting function w we can set up the weighted integral

$$\nabla^2 t = 0 \Rightarrow \int_\Omega w \nabla^2 t d\Omega = 0$$

Applying the Green-Gauss theorem once

$$\int_{\Gamma_1 \cup \Gamma_2} \frac{dt}{dn} w d\Gamma - \int_\Omega \nabla t \nabla w d\Omega = 0$$

and a second time on the last integral yields

$$\int_{\Gamma_1 \cup \Gamma_2} \frac{dt}{dn} w d\Gamma - \int_{\Gamma_1 \cup \Gamma_2} t \frac{dw}{dn} d\Gamma + \int_\Omega t \nabla^2 w d\Omega = 0. \tag{D.2}$$

A fundamental function (or Green's function) is a function solving a PDE with a point source and no boundary conditions. In our case the fundamental solution to $\nabla^2 t = \delta(x_0 - x, y_0 - y)$ where δ is the usual Dirac's delta function is

$$w(x, y) = -\frac{1}{2\pi} \log r, \text{ where } r = \sqrt{(x_0 - x)^2 + (y_0 - y)^2}.$$

We now choose the weight function w to be the fundamental solution and inserting into the last integral of (D.2) gives

$$\int_\Omega t \nabla^2 w d\Omega = -\int_\Omega t \delta(x_0 - x, y_0 - y) d\Omega = -t(x_0, y_0).$$

That is, by this choice of w we have replaced the domain integral by a point value, and inserted into Eq. (D.2) we get

$$t(x_0, y_0) + \int_{\Gamma_1 \cup \Gamma_2} t \frac{dw}{dn} d\Gamma = \int_{\Gamma_1 \cup \Gamma_2} \frac{dt}{dn} w d\Gamma,$$

and we have only integrals of the boundary type left. If we divide the boundaries into small pieces called elements and assume that each element Γ_j has constant temperature t_j and temperature gradient dt_j/dt we get the linear equation

$$t(x_0, y_0) + \sum_j t_j \int_{\Gamma_j} \frac{dw}{dn} d\Gamma = \sum_j \frac{dt_j}{dn} \int_{\Gamma_j} w d\Gamma. \tag{D.3}$$

If we select (x_0, y_0) to be the midpoint of each element in turn we get N equations

$$(\mathbf{G} + \mathbf{I})\mathbf{t} = \mathbf{H}\mathbf{u},$$

where the matrices are

$$G_{ij} = \int_{\Gamma_j} \frac{dw}{dn}(x_i, y_i) d\Gamma, \; H_{ij} = \int_{\Gamma_j} w(x_i, y_i) d\Gamma$$

and the vectors are

$$\mathbf{t} = [t_1\, t_2\, \ldots t_n]^T \text{ and } \mathbf{u} = [dt_1/dn\, dt_2/dn\, \ldots\, dt_n/dn]^T.$$

Depending on the boundary conditions some elements $\mathbf{t}$ and $\mathbf{u}$ are known and others are unknown.

In our case either t_i or $u_i = dt_i/dn$ is known and we see that we have N equations with N unknowns.

If we need the solution at a point (x, y) not on the boundary we use the sum in (D.3).

The SteadyHeatBEM class uses BEM to solve the problem. The integrals involving the fundamental solution are computed via the quadrature method in the quadg function in Matlab. This presentation of the most simple BEM is partly based on the excellent introduction in [16] where more advanced methods and PDEs are treated.

D.2 Interpolation

The interpolation problem is inspired by the inverse interpolation problem used in [113]. Instead of a real world measured data set it uses randomly generated points. The forward problem interpolates from a regular grid to a number of irregular points.

The problem implements bilinear interpolation, that is, if we want to interpolate $f(x, y)$ from the $f(x_0, y_0)$, $f(x_0, y_1)$, $f(x_1, y_0)$ and $f(x_1, y_1)$, see Fig. D.2 , we use the formula

$$\begin{aligned} f(x,y) = \Big(& f(x_0,y_0)(x_1-x)(y_1-y) + \\ & f(x_0,y_1)(x_1-x)(y-y_0) + \\ & f(x_1,y_0)(x-x_0)(y_1-y) + \\ & f(x_1,y_1)(x-x_0)(y-y_0)\Big) / \Big((x_1-x_0)(y_1-y_0)\Big) \end{aligned}$$

With n points in the regular grid and m points in the irregular grid we arrive at an operator $\mathbf{K} \in \mathbb{R}^{m\times n}$. The actual operator implements the multiplication with a C mex-function illustrating how code in other languages can be used in connection with the toolbox. The implementation is in the Interpolate class.

D.3 Deblurring Problems

The well known BLUR test example has been modified to take advantage of the KroneckerProduct2D class. Hereby it is possible to work with bigger problems than before as the $n^2 \times n^2$ matrix is not created explicitly. Due to the Kronecker product structure the SVD is available and is used in the TSVD method (of course), Tikhonov and several other functions.

Furthermore, a couple of variations of blur using KroneckerProduct2D are included, see Table D.1.

Finally the classes psf, psfMatrix and psfPrec from Restore Tools by Nagy et al. [100] are included. They handle non-separable and spatial invariant blur. Furthermore, it is possible to specify boundary conditions. The original versions did not

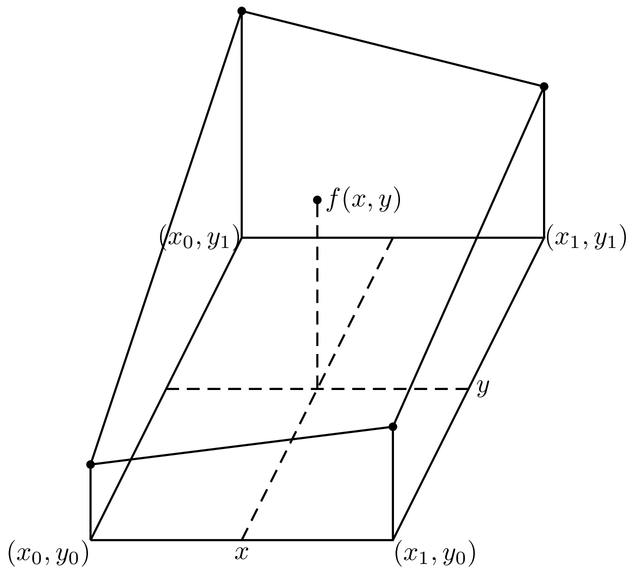


Figure D.2: Geometry of interpolation

Name	Kernel K	Description
BLUR	$\frac{1}{\pi\sigma}\exp\left(-\frac{(x-s)^2+(y-t)^2}{\sigma^2}\right)$	Atmospheric blur
OBLUR	$\begin{cases} 1/(\pi R^2)), & \text{if } \sqrt{(x-s)^2+(y-t)^2} \leq R \\ 0, & \text{else} \end{cases}$	Out-of-focus blur
MBLUR	$\begin{cases} 1/(2L), & \text{if } \lvert x-s\rvert \leq L \\ 0, & \text{else} \end{cases}$	Motion blur
CONFOCAL	$d_x d_y \text{sinc}(\pi d_x(x-s))\text{sinc}(\pi d_y(y-t))$	Confocal blur

Table D.1: Blur test problems. All functions create a KroneckerProduct2D object with terms found from the discretization of a integral equation of the form $\int\int K(x,y,s,t)f(x,y)dxdy = g(s,t)$. The scalars L, d and R describe the extent of the blur and can be varied to control the level of ill-conditioning of the problem.

inherit from LinearOperator and did not use object oriented techniques for the vectors. In this package, they are changed to inherit properly; see the object constructors. In methods where vectors are used, for example, in the multiplication operation, we first unpack the data of a Vector object, then we do the operation and finally we wrap the result into the proper type of Vector. The changes are clearly marked by comments in the code.

D.4 Gravity

The GRAVITY test problem described in [126] is a one dimensional Fredholm integral equation of the first kind,

$$g(s) = \int_0^1 \frac{d}{\sqrt{(d^2 + (s-t)^2)^3}} f(t)dt,$$

where d is the depth of the mass-distribution $f(t)$ we want to find from the measurements $g(s)$ at the surface. Larger depth d yields a more ill-posed problem. The GRAVITY test problem is implemented via the Matrix class.

Appendix E

Subspace Preconditioned LSQR for Discrete Ill-Posed Problems

The following pages contain the paper now publicized in BIT [71]. The work is based on my Master's thesis [81], in turn based on work by Hanke and Vogel [62]. Michael Saunders suggested a significant improvement to the algorithm based on LSQR and the result is presented in the following paper. The paper is coauthored by Per Christian Hansen, Michael Jacobsen and Michael Saunders.

How the algorithm interacts with object oriented ideas and in particular the situations where the current implementation does not work is described in Sec. 4.6.1.

BIT
2003, Vol. xx, No. x, pp. 001–016

0006-3835/00/4004-0001 $15.00

SUBSPACE PRECONDITIONED LSQR FOR DISCRETE ILL-POSED PROBLEMS *

M. JACOBSEN[1], P. C. HANSEN[1] and M. A. SAUNDERS[2] †

[1] *Informatics and Mathematical Modelling, Technical University of Denmark, Building 321, DK-2800 Kgs. Lyngby, Denmark. email: {mj, pch}@imm.dtu.dk*

[2] *Department of Management Science and Engineering, Terman Building, Stanford University, Stanford, CA 94305-4026, USA. email: saunders@stanford.edu*

Abstract.

We present a novel implementation of a two-level iterative method for the solution of discrete linear ill-posed problems. The algorithm is algebraically equivalent to the two-level Schur complement CG algorithm of Hanke and Vogel, but involves less work per iteration. We review the algorithm, discuss our implementation, and show promising results from numerical experiments that give insight into the proper use of the algorithm.

AMS subject classification: 65F10, 65F22

Key words: ill-posed problems, regularization, two-level iterative methods, Schur complement CG method, LSQR

1 Introduction.

Linear ill-posed problems arise in a variety of applications in science and engineering, and reliable computational algorithms are available for small- and medium-scale problems [6, 18]. However, there is a need for iterative algorithms that can treat large-scale problems, i.e., those with sparse or structured matrices, or where the action of the linear operator is provided by some computational scheme.

This work focuses on a large-scale iterative algorithm for computing solutions to the Tikhonov regularization problem

$$\min_{\mathbf{x}} \; \|\mathbf{K}\mathbf{x} - \mathbf{y}\|_2^2 + \lambda^2 \|\mathbf{L}\mathbf{x}\|_2^2,$$

in which $\mathbf{K} \in \mathbb{R}^{m\times n}$ is a given matrix, $\mathbf{L} \in \mathbb{R}^{p\times n}$ is a matrix that defines a suitable smoothing norm for the problem, $\mathbf{y} \in \mathbb{R}^m$ is a vector of observed data, and the regularization parameter λ controls the weight between the residual and

*Received xxx. Revised xxx. Communicated by Zdeněk Strakoš.

†Partially supported by Danish Natural Science Research Foundation grant 9901581, U.S. National Science Foundation grant CCR-9988205, and U.S. Office of Naval Research grants N00014-96-1-0274 and N00014-02-1-0076. Version of February 25, 2003

smoothing norms. We note that both $\mathbf{K}$ and $\mathbf{L}$ may be available either explicitly (as sparse or structured matrices) or implicitly through their action on a vector.

We present an efficient implementation of an iterative algorithm developed by Hanke and Vogel [5], who discussed several "two-level methods" for the solution of the Tikhonov problem. The algorithm is based on the least squares formulation of the Tikhonov problem,

$$\mathbf{x}_\lambda = \underset{\mathbf{x}}{\operatorname{argmin}} \left\| \widehat{\mathbf{K}}\mathbf{x} - \widehat{\mathbf{y}} \right\|_2, \tag{1.1}$$

where we introduce the "stacked" matrix and vector

$$\widehat{\mathbf{K}} = \begin{bmatrix} \mathbf{K} \\ \lambda \mathbf{L} \end{bmatrix}, \qquad \widehat{\mathbf{y}} = \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix}.$$

The key idea in the work of Hanke and Vogel is a splitting of the solution space $\mathbb{R}^n$ into two subspaces, one of them with a small dimension and with basis vectors chosen such that this subspace represents approximate regularized solutions. Similarly to multi-level methods, the residual is projected into a small-dimensional subspace (the coarse grid) in which the problem is solved with a direct method, while the component of the solution in the remaining subspace is computed by an iterative algorithm. Because of this analogy, the methods are denoted two-level methods. The obvious extension to a multi-level method was tested but without success [17].

Two of the three methods in [5] are based on the preconditioning idea for the conjugate gradient (CG) method [3, 8], but we do not treat these preconditioning methods here because the third method performs better, in theory and in practice [5, 9].

The third method in [5]—the only one considered here—is based on the Schur complement CG algorithm; see Axelsson [2]. When the subspace splitting idea is combined with the Schur complement CG algorithm, one obtains a method in which a special Schur complement system is solved iteratively by means of CG.

Our main contribution is to show that this Schur complement system is, in fact, the normal equations for an underlying least squares problem, and to demonstrate how this problem can be solved efficiently by means of the LSQR algorithm [12, 13]. There are two advantages in this approach: the implementation is more efficient, and the algorithm is numerically more robust because it avoids the use of the normal equations. (We focus on the use of LSQR because of its superior stability, and because we want to give all the details of the implementation. Any similar iterative method for least squares problems could be used, such as CGLS [8, 12], and the efficiency would be the same.)

In addition, we discuss various strategies for choosing the basis vectors of the subspace with small dimension, and we demonstrate by examples how this choice affects the convergence of the method.

We first summarize the derivation of the Schur CG algorithm from [5] in §2. Next, in §3 we show how an equivalent method can be obtained by means of a certain QR factorization, with the advantage of a simpler algorithm with fewer operations per iteration. Then in §4 we discuss various details on how to select the subspace splitting, and in §5 we show numerical tests on two test problems.

2 Two-Level Schur Complement CG.

For the presentation of the Hanke-Vogel algorithm it is convenient to introduce the quantities

$$\mathbf{A} = \mathbf{K}^{\mathrm{T}}\mathbf{K} + \lambda^2\mathbf{L}^{\mathrm{T}}\mathbf{L} = \widehat{\mathbf{K}}^{\mathrm{T}}\widehat{\mathbf{K}}, \qquad \mathbf{b} = \mathbf{K}^{\mathrm{T}}\mathbf{y} = \widehat{\mathbf{K}}^{\mathrm{T}}\widehat{\mathbf{y}}$$

associated with the normal equations for the least squares problem (1.1). For ease of presentation we assume in this section that the matrix $\mathbf{L}$ has full column rank; otherwise the algorithm becomes substantially more complicated [15].

The two levels are created by a splitting of the solution space $\mathbb{R}^n$ into two subspaces $\mathcal{V}$ and $\mathcal{W}$ of dimensions k and $n-k$, respectively. To make the algorithm practical one should choose $k \ll n$. Let the columns of the matrix $\mathbf{V} \in \mathbb{R}^{n\times k}$ span the subspace $\mathcal{V}$, and let the columns of the matrix $\mathbf{W} \in \mathbb{R}^{n\times(n-k)}$ span the subspace $\mathcal{W}$. The solution is obtained in the form

$$\mathbf{x}_\lambda = \mathbf{V}\mathbf{v} + \mathbf{W}\mathbf{w}. \tag{2.1}$$

The columns of $\mathbf{V}$ are not required to be orthonormal, although this is preferable in a numerical implementation. As we shall see, the matrix $\mathbf{W}$ is not needed in the algorithm—it is only necessary for its derivation.

Hanke and Vogel choose the two subspaces to be $\mathbf{L}^{\mathrm{T}}\mathbf{L}$-orthogonal, and the columns of $\mathbf{W}$ to be $\mathbf{L}^{\mathrm{T}}\mathbf{L}$-orthonormal. Thus,

$$\mathbf{V}^{\mathrm{T}}\mathbf{L}^{\mathrm{T}}\mathbf{L}\mathbf{W} = \mathbf{0} \quad \text{and} \quad \mathbf{W}^{\mathrm{T}}\mathbf{L}^{\mathrm{T}}\mathbf{L}\mathbf{W} = \mathbf{I}.$$

These properties are not needed for the final algorithm as presented below, but they are necessary for the omitted details in the derivation of the algorithm.

First we transform the normal equations $\mathbf{A}\mathbf{x}_\lambda = \mathbf{b}$ to the equivalent form

$$[\,\mathbf{V}\ \mathbf{W}\,]^{\mathrm{T}}\,\mathbf{A}\,[\,\mathbf{V}\ \mathbf{W}\,]\begin{bmatrix}\mathbf{v}\\ \mathbf{w}\end{bmatrix} = [\,\mathbf{V}\ \mathbf{W}\,]^{\mathrm{T}}\,\mathbf{b}, \tag{2.2}$$

which expands into the block system

$$\begin{bmatrix}\mathbf{A}_{11} & \mathbf{V}^{\mathrm{T}}\mathbf{A}\mathbf{W}\\ \mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{V} & \mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{W}\end{bmatrix}\begin{bmatrix}\mathbf{v}\\ \mathbf{w}\end{bmatrix} = \begin{bmatrix}\mathbf{V}^{\mathrm{T}}\mathbf{b}\\ \mathbf{W}^{\mathrm{T}}\mathbf{b}\end{bmatrix}, \tag{2.3}$$

where $\mathbf{A}_{11} = \mathbf{V}^{\mathrm{T}}\mathbf{A}\mathbf{V} \in \mathbb{R}^{k\times k}$. The algorithm in its final form requires $\mathbf{A}_{11}$, and otherwise only uses $\mathbf{K}$, $\mathbf{L}$, $\mathbf{b}$ and $\mathbf{V}$.

We now apply block Gaussian elimination to (2.3) as in [2] to obtain

$$\begin{bmatrix}\mathbf{A}_{11} & \mathbf{V}^{\mathrm{T}}\mathbf{A}\mathbf{W}\\ \mathbf{0} & \mathbf{S}\end{bmatrix}\begin{bmatrix}\mathbf{v}\\ \mathbf{w}\end{bmatrix} = \begin{bmatrix}\mathbf{V}^{\mathrm{T}}\mathbf{b}\\ \mathbf{s}\end{bmatrix}$$

with

$$\begin{aligned}\mathbf{S} &= \mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{W} - \mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{V}\mathbf{A}_{11}^{-1}\mathbf{V}^{\mathrm{T}}\mathbf{A}\mathbf{W},\\ \mathbf{s} &= \mathbf{W}^{\mathrm{T}}\mathbf{b} - \mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{V}\mathbf{A}_{11}^{-1}\mathbf{V}^{\mathrm{T}}\mathbf{b},\end{aligned}$$

where $\mathbf{S}$ is the Schur complement of $\mathbf{A}_{11}$ in the matrix (2.3). We see that $\mathbf{x}_\lambda$ is obtained from

$$\mathbf{S}\mathbf{w} = \mathbf{s}, \tag{2.4}$$

$$\mathbf{A}_{11}\mathbf{v} = \mathbf{V}^{\mathrm{T}}(\mathbf{b} - \mathbf{A}\mathbf{W}\mathbf{w}), \tag{2.5}$$

$$\mathbf{x}_\lambda = \mathbf{V}\mathbf{v} + \mathbf{W}\mathbf{w}. \tag{2.6}$$

The Schur complement $\mathbf{S} \in \mathbb{R}^{(n-k)\times(n-k)}$ is symmetric positive definite because $\mathbf{A}$ is so [2, Theorem 3.9], and therefore CG comes to mind as a choice for solving the Schur system (2.4) when $n-k$ is large. The resulting algorithm (see below) is expressed in terms of the original matrices $\mathbf{K}$ and $\mathbf{L}$ only. The multiplication with $\mathbf{V}\mathbf{A}_{11}^{-1}\mathbf{V}^{\mathrm{T}}$ can be considered a coarse grid correction step. The rather lengthy details of this derivation are omitted here, but they are available in [5] and [9].

Schur complement CG. Computes the Tikhonov regularized solution $\mathbf{x}_\lambda = \operatorname{argmin}_{\mathbf{x}} \|\widehat{\mathbf{K}}\mathbf{x} - \widehat{\mathbf{y}}\|_2$. The product $\mathbf{L}^{\mathrm{T}}\mathbf{L}$ must be invertible.

1: $\mathbf{x}_0 = \mathbf{V}\mathbf{A}_{11}^{-1}\mathbf{V}^{\mathrm{T}}\mathbf{b}$
2: $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
3: $\mathbf{y}_0 = (\mathbf{L}^{\mathrm{T}}\mathbf{L})^{-1}\mathbf{r}_0$
4: $\mathbf{d}_0 = \mathbf{r}_0$
5: $\mathbf{z}_0 = \mathbf{y}_0$
6: $\rho_0 = (\mathbf{y}_0)^{\mathrm{T}}\mathbf{r}_0$
7: $i = 0$
8: **repeat**
9: $\quad \mathbf{v}_i = (\mathbf{I} - \mathbf{V}\mathbf{A}_{11}^{-1}\mathbf{V}^{\mathrm{T}}\mathbf{A})\,\mathbf{z}_i$
10: $\quad \mathbf{w}_i = \mathbf{A}\mathbf{v}_i$
11: $\quad \mathbf{g}_i = (\mathbf{L}^{\mathrm{T}}\mathbf{L})^{-1}\mathbf{w}_i$
12: $\quad \beta_i = \rho_i/(\mathbf{d}_i)^{\mathrm{T}}\mathbf{g}_i$
13: $\quad \mathbf{x}_{i+1} = \mathbf{x}_i + \beta_i\mathbf{v}_i$
14: $\quad \mathbf{r}_{i+1} = \mathbf{r}_i - \beta_i\mathbf{w}_i$
15: $\quad \mathbf{y}_{i+1} = \mathbf{y}_i - \beta_i\mathbf{g}_i$
16: $\quad \rho_{i+1} = (\mathbf{y}_{i+1})^{\mathrm{T}}\mathbf{r}_{i+1}$
17: $\quad \gamma_{i+1} = \rho_{i+1}/\rho_i$
18: $\quad \mathbf{d}_{i+1} = \mathbf{r}_{i+1} + \gamma_{i+1}\mathbf{d}_i$
19: $\quad \mathbf{z}_{i+1} = \mathbf{y}_{i+1} + \gamma_{i+1}\mathbf{z}_i$
20: $\quad i = i + 1$
21: **until** $\|\mathbf{r}_i\|_2 > \|\mathbf{r}_{i-1}\|_2$ or $i =$ iteration limit
22: $\mathbf{x}_\lambda = \mathbf{x}_i$

Each iteration involves solving one system with $\mathbf{A}_{11}$ (in step 9) and one with $\mathbf{L}^{\mathrm{T}}\mathbf{L}$ (in step 11). The Cholesky factors of these matrices are the triangular QR factors of $\widehat{\mathbf{K}}\mathbf{V}$ and $\mathbf{L}$, respectively. If $\mathbf{L}$ is large and sparse then the solution in step 11 can also be computed via an iterative method [14].

Note that one iteration also requires two applications of $\mathbf{A}$ (i.e., two applications of $\widehat{\mathbf{K}}$ and $\widehat{\mathbf{K}}^{\mathrm{T}}$ each) and an application of both $\mathbf{V}$ and $\mathbf{V}^{\mathrm{T}}$. Hence one iteration is at least twice as expensive as one iteration of LSQR or CGLS applied to the original least squares problem (1.1), and the rate of convergence for the two-level method must therefore be at least twice that of LSQR and CGLS for the two-level method to be a better choice.

The stopping criterion is based on the decrease of the residual norm. Since the residual is given in terms of the original system and not the Schur complement a "breakdown" occurs when the Schur complement residual reaches the machine precision. We observed that the breakdown destroys the assumed relationship between the updated vectors, and the method diverges for the next couple of iterations until the relationships are reestablished.

3 The Subspace Preconditioned LSQR Algorithm.

It is well known that the LSQR algorithm is algebraically identical to CGLS and to the CG algorithm on the normal equation system. However, LSQR can be considered as treating the least squares problem in a more direct way, and in finite precision LSQR is often capable of achieving more accurate results than CGLS and in particular CG. Therefore we wish to find the LSQR equivalent of the Schur complement CG algorithm.

As before we want a solution of the form $\mathbf{x}_\lambda = \mathbf{V}\mathbf{v} + \mathbf{W}\mathbf{w}$ to the least squares system (1.1). Thus we need to obtain

$$\begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix} = \operatorname*{argmin}_{\mathbf{v},\mathbf{w}} \left\| \widehat{\mathbf{K}} \, [\, \mathbf{V} \; \mathbf{W} \,] \begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix} - \widehat{\mathbf{y}} \right\|_2 .$$

Introducing the QR factorization

$$\widehat{\mathbf{K}}\mathbf{V} = \mathbf{Q}\widehat{\mathbf{R}} = [\, \mathbf{Y} \; \mathbf{Z} \,] \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} = \mathbf{Y}\mathbf{R} \tag{3.1}$$

with $\mathbf{Q}$ orthogonal and $\mathbf{R}$ upper triangular, we get

$$\begin{aligned}
\begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix} &= \operatorname*{argmin}_{\mathbf{v},\mathbf{w}} \left\| \mathbf{Q}^{\mathrm{T}}\widehat{\mathbf{K}} \, [\, \mathbf{V} \; \mathbf{W} \,] \begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix} - \mathbf{Q}^{\mathrm{T}}\widehat{\mathbf{y}} \right\|_2 \\
&= \operatorname*{argmin}_{\mathbf{v},\mathbf{w}} \left\| \begin{bmatrix} \mathbf{Y}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{V} & \mathbf{Y}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{W} \\ \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{V} & \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{W} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix} - \begin{bmatrix} \mathbf{Y}^{\mathrm{T}}\widehat{\mathbf{y}} \\ \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{y}} \end{bmatrix} \right\|_2 \\
&= \operatorname*{argmin}_{\mathbf{v},\mathbf{w}} \left\| \begin{bmatrix} \mathbf{R} & \mathbf{Y}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{W} \\ \mathbf{0} & \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{W} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix} - \begin{bmatrix} \mathbf{Y}^{\mathrm{T}}\widehat{\mathbf{y}} \\ \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{y}} \end{bmatrix} \right\|_2 .
\end{aligned}$$

Thus, the Tikhonov solution is found from

$$\mathbf{w} = \operatorname*{argmin}_{\mathbf{w}} \left\| \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{W}\mathbf{w} - \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{y}} \right\|_2, \tag{3.2}$$

$$\mathbf{R}\mathbf{v} = \mathbf{Y}^{\mathrm{T}}\big(\widehat{\mathbf{y}} - \widehat{\mathbf{K}}\mathbf{W}\mathbf{w}\big), \tag{3.3}$$

$$\mathbf{x}_\lambda = \mathbf{V}\mathbf{v} + \mathbf{W}\mathbf{w}. \tag{3.4}$$

The partial QR factors of $\widehat{\mathbf{K}}[\,\mathbf{V}\ \mathbf{W}\,]$ give $[\,\mathbf{R}\ \ \mathbf{Y}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{W}\,]$, which would be the top part of a full QR factorization. After $\mathbf{w}$ is determined, $\mathbf{v}$ is found as if one were continuing back-substitution with the full $\mathbf{R}$ factor in the normal way.

The QR factors in (3.1) can be computed conveniently by means of Householder transformations. Since the number of columns k in the matrix product $\widehat{\mathbf{K}}\mathbf{V}$ is small compared to the dimensions of the problem, the k Householder vectors efficiently represent the orthogonal matrix $\mathbf{Q} = [\,\mathbf{Y}\ \mathbf{Z}\,]$ in product form. Then products of the form $\mathbf{Zu}$ and $\mathbf{Z}^{\mathrm{T}}\mathbf{v}$ are done by actually multiplying with $\mathbf{Q}$ and $\mathbf{Q}^{\mathrm{T}}$, which in turn uses the stored Householder transformations, requiring about $4k(m+p)$ flops per product.

3.1 Equivalence.

We now show that the Schur complement system $\mathbf{Sw} = \mathbf{s}$ (2.4) is indeed the normal equations of the least squares system (3.2). First note from the QR factorization of $\widehat{\mathbf{K}}\mathbf{V}$ that $\mathbf{A}_{11} = \mathbf{V}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{V} = \mathbf{R}^{\mathrm{T}}\mathbf{R}$. Hence

$$
\begin{aligned}
\mathbf{S} &= \mathbf{W}^{\mathrm{T}}(\mathbf{A} - \mathbf{AVA}_{11}^{-1}\mathbf{V}^{\mathrm{T}}\mathbf{A})\mathbf{W} \\
&= \mathbf{W}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\left(\mathbf{I} - \widehat{\mathbf{K}}\mathbf{VA}_{11}^{-1}\mathbf{V}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\right)\widehat{\mathbf{K}}\mathbf{W} \\
&= \mathbf{W}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\left(\mathbf{I} - \mathbf{YR}(\mathbf{R}^{\mathrm{T}}\mathbf{R})^{-1}\mathbf{R}^{\mathrm{T}}\mathbf{Y}^{\mathrm{T}}\right)\widehat{\mathbf{K}}\mathbf{W} \\
&= \mathbf{W}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}(\mathbf{I} - \mathbf{YY}^{\mathrm{T}})\widehat{\mathbf{K}}\mathbf{W} \\
&= \mathbf{W}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\mathbf{ZZ}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{W},
\end{aligned}
$$

and

$$
\begin{aligned}
\mathbf{s} &= \mathbf{W}^{\mathrm{T}}\left(\mathbf{I} - \mathbf{AVA}_{11}^{-1}\mathbf{V}^{\mathrm{T}}\right)\mathbf{b} \\
&= \mathbf{W}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\left(\mathbf{I} - \widehat{\mathbf{K}}\mathbf{VA}_{11}^{-1}\mathbf{V}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\right)\widehat{\mathbf{y}} \\
&= \mathbf{W}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\mathbf{ZZ}^{\mathrm{T}}\widehat{\mathbf{y}},
\end{aligned}
$$

as required. Hence CG applied to the Schur system (2.4) and LSQR or CGLS applied to the least squares system (3.2) produce the same results in infinite precision. In finite precision LSQR usually yields results with slightly higher precision.

We can also show that systems (2.5) and (3.3) are both equivalent to the least squares problem $\min_{\mathbf{v}} \|\widehat{\mathbf{K}}\mathbf{V}\mathbf{v} - (\widehat{\mathbf{y}} - \widehat{\mathbf{K}}\mathbf{W}\mathbf{w})\|_2$. The Schur complement approach uses the associated normal equations to solve for $\mathbf{v}$, while our approach uses the QR factorization (3.1).

More important, however, is that the QR approach leads to a simpler iteration scheme and thus a simpler implementation. In particular, in each iteration we only need one multiplication with $\widehat{\mathbf{K}}$ and $\widehat{\mathbf{K}}^{\mathrm{T}}$ (while the Schur complement CG algorithm needs two multiplications). Moreover, there is no linear equation system involving $\mathbf{L}^{\mathrm{T}}\mathbf{L}$, and there is no requirement on the rank of $\mathbf{L}$ (as long as $\widehat{\mathbf{K}}$ has full rank).

3.2 Solving for $\mathbf{Ww}$ *and* $\mathbf{v}$.

Defining $\mathbf{p} = \mathbf{Ww}$, we first show how to proceed in cases where $\mathbf{W}$ is not available. Substituting into (3.2)–(3.4) gives the equations

$$\mathbf{p} = \operatorname*{argmin}_{\mathbf{p}} \left\| \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{p} - \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{y}} \right\|_2, \tag{3.5}$$

$$\mathbf{Rv} = \mathbf{Y}^{\mathrm{T}}\big(\widehat{\mathbf{y}} - \widehat{\mathbf{K}}\mathbf{p}\big), \tag{3.6}$$

$$\mathbf{x}_\lambda = \mathbf{Vv} + \mathbf{p}, \tag{3.7}$$

in which the main computation is solving for $\mathbf{p}$. The resulting algorithm is displayed below, where for completeness all details of the LSQR iterations are included. The Lanczos vectors are denoted by $\bar{\mathbf{u}}_i$ and $\bar{\mathbf{v}}_i$, and the $\bar{\mathbf{w}}_i$ are work vectors.

Subspace preconditioned LSQR. Computes the Tikhonov regularized solution $\mathbf{x}_\lambda = \operatorname{argmin}_{\mathbf{x}} \|\widehat{\mathbf{K}}\mathbf{x} - \widehat{\mathbf{y}}\|_2$. There is no restriction on the rank of $\mathbf{L}$.

Compute QR factorization $\widehat{\mathbf{K}}\mathbf{V} = [\,\mathbf{Y}\ \mathbf{Z}\,]\begin{bmatrix}\mathbf{R}\\ \mathbf{0}\end{bmatrix}$
$\mathbf{d} = \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{y}}$
$\beta_1\bar{\mathbf{u}}_1 = \mathbf{d}$ % So that $\|\bar{\mathbf{u}}_1\|_2 = 1$
$\alpha_1\bar{\mathbf{v}}_1 = \widehat{\mathbf{K}}^{\mathrm{T}}\mathbf{Z}\bar{\mathbf{u}}_1$ % So that $\|\bar{\mathbf{v}}_1\|_2 = 1$
$\bar{\mathbf{w}}_1 = \bar{\mathbf{v}}_1$
$\mathbf{p}_0 = \mathbf{0}$
$\bar{\phi}_1 = \beta_1$
$\bar{\rho}_1 = \alpha_1$
$i = 0$
repeat
    $\beta_{i+1}\bar{\mathbf{u}}_{i+1} = \mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}\bar{\mathbf{v}}_i - \alpha_i\bar{\mathbf{u}}_i$ % So that $\|\bar{\mathbf{u}}_{i+1}\|_2 = 1$
    $\alpha_{i+1}\bar{\mathbf{v}}_{i+1} = \widehat{\mathbf{K}}^{\mathrm{T}}\mathbf{Z}\bar{\mathbf{u}}_{i+1} - \beta_{i+1}\bar{\mathbf{v}}_i$ % So that $\|\bar{\mathbf{v}}_{i+1}\|_2 = 1$
    $\rho_i = (\bar{\rho}_i^2 + \beta_{i+1}^2)^{1/2}$
    $c_i = \bar{\rho}_i/\rho_i$
    $s_i = \beta_{i+1}/\rho_i$
    $\theta_{i+1} = s_i\alpha_{i+1}$
    $\bar{\rho}_{i+1} = -c_i\alpha_{i+1}$
    $\phi_i = c_i\bar{\phi}_i$
    $\bar{\phi}_{i+1} = s_i\bar{\phi}_i$
    $\mathbf{p}_i = \mathbf{p}_{i-1} + (\phi_i/\rho_i)\bar{\mathbf{w}}_i$
    $\bar{\mathbf{w}}_{i+1} = \bar{\mathbf{v}}_{i+1} - (\theta_{i+1}/\rho_i)\bar{\mathbf{w}}_i$
    $i \leftarrow i + 1$
until $\bar{\phi}_i\alpha_i|c_{i-1}| < \tau$
Solve $\mathbf{Rv} = \mathbf{Y}^{\mathrm{T}}(\widehat{\mathbf{y}} - \widehat{\mathbf{K}}\mathbf{p}_{i-1})$ for $\mathbf{v}$
$\mathbf{x}_\lambda = \mathbf{Vv} + \mathbf{p}_{i-1}$

Observe that $\mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{V} = \mathbf{Z}^{\mathrm{T}}\mathbf{YR} = \mathbf{0}$, and therefore by construction, the matrix $\mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}$ has a non-trivial null space spanned by the columns of $\mathbf{V}$. For this reason

LSQR will eventually diverge as a result of finite precision effects, cf. [12, §6.2]. Our experiments agree with those in [12], leading to a stopping criterion based on the approximation $\bar{\phi}\alpha|c| \approx \|\widehat{\mathbf{K}}^{\mathrm{T}}\mathbf{Z}\mathbf{r}\|_2 < \tau$, where $\mathbf{r} = \mathbf{Z}^{\mathrm{T}}(\widehat{\mathbf{y}} - \widehat{\mathbf{K}}\mathbf{p})$ is the residual for the least squares problem (3.2), and τ is a tolerance of order 10^{-12}, for example. This stopping criterion ends the iterations just before divergence sets in.

3.3 Solving for $\mathbf{w}$ *and* $\mathbf{v}$.

The difficulties with the rank deficiency in $\mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}$ can be avoided by returning to (3.2)–(3.4) and working with the slightly smaller vector $\mathbf{w}$ and matrix $\mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{W}$. The following lines must be changed:

4: $\alpha_1\bar{\mathbf{v}}_1 = \mathbf{W}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\mathbf{Z}\bar{\mathbf{u}}_1$
6: $\mathbf{w}_0 = \mathbf{0}$
11: $\beta_{i+1} = \bar{\mathbf{u}}_{i+1}\mathbf{Z}^{\mathrm{T}}\widehat{\mathbf{K}}\mathbf{W}\bar{\mathbf{v}}_i - \alpha_i\bar{\mathbf{u}}_i$
12: $\alpha_{i+1} = \bar{\mathbf{v}}_{i+1}\mathbf{W}^{\mathrm{T}}\widehat{\mathbf{K}}^{\mathrm{T}}\mathbf{Z}\bar{\mathbf{u}}_{i+1} - \beta_{i+1}\bar{\mathbf{v}}_i$
20: $\mathbf{w}_i = \mathbf{w}_{i-1} + (\phi_i/\rho_i)\bar{\mathbf{w}}_i$
24: Solve $\mathbf{R}\mathbf{v} = \mathbf{Y}^{\mathrm{T}}(\widehat{\mathbf{y}} - \widehat{\mathbf{K}}\mathbf{W}\mathbf{w}_{i-1})$ for $\mathbf{v}$
25: $\mathbf{x}_\lambda = \mathbf{V}\mathbf{v} + \mathbf{W}\mathbf{w}_{i-1}$

This approach is practical only if operations with $\mathbf{W}$ and $\mathbf{W}^{\mathrm{T}}$ can be done quickly.

4 The Subspace Splitting.

The optimal subspace $\mathcal{V}$ consists of the principal k right singular vectors in the SVD of $\mathbf{K}$ if $\mathbf{L} = \mathbf{I}$, or generalized right singular vectors in the GSVD of $(\mathbf{K}, \mathbf{L})$ if $\mathbf{L} \neq \mathbf{I}$. This choice diagonalizes the submatrix $\mathbf{A}_{11}$ and minimizes the condition number of $\mathbf{S}$, which indicates faster convergence of the algorithms. However, the optimal subspace created from the SVD or GSVD is usually not available, and we are forced to choose a less optimal subspace. The hope is that a subspace "close to" the optimal (G)SVD-based subspace is good enough. We know that in most applications the right singular vectors of an ill-posed problem have increasingly more sign changes as the corresponding (generalized) singular values decrease, i.e., they become more oscillatory—a feature we should try to emulate.

In the previous sections the subspace operations were described by means of matrix multiplications with $\mathbf{V}$ and $\mathbf{V}^{\mathrm{T}}$. However, if we select a specific subspace $\mathcal{V}$ then the multiplications can often be done more cheaply by means of "fast transforms." Two examples come to mind.

- The wavelet transform; in this paper we use the Daubechies wavelets [4].
- The Fourier transform and variations thereof; to avoid complex results we restrict ourself to the use of the cosine transform denoted DCT-2 in [16].

Both transforms have the desired property that the basis vectors have higher frequency as the column index grows. This is obviously true for the cosine

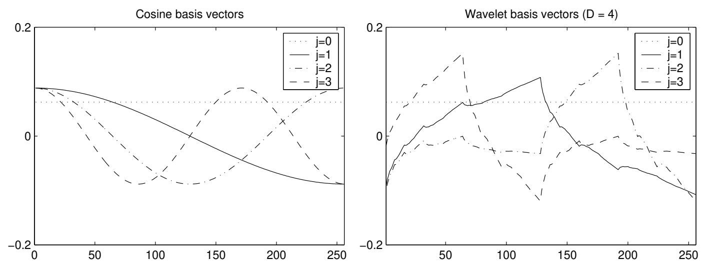


Figure 4.1: The first four basis vectors for DCT-2 (left) and wavelets (right).

transform, which is spanned by the vectors

$$\mathbf{V}_{i,j} = \left(\frac{n}{2}\right)^{-1/2} \cos\left((i+1/2)(j-1)\frac{\pi}{n}\right), \quad \text{divided by } \sqrt{2} \text{ if } j=1. \tag{4.1}$$

The Daubechie wavelets, on the other hand, have a local nature and oscillate only in their domain of support. The wavelet basis is composed of several subspaces of increasing detail,

$$\mathbb{R}^n = \mathcal{V}_0 \bigoplus_{j=1}^{k} \mathcal{W}_j \ , \quad k = \log_2 n.$$

Because all vectors spanning $\mathcal{W}_j$ must be included to describe all details at the given level, the subspace dimension k should preferably be a power of 2. Figure 4.1 shows the first four basis vectors of the two fast transforms.

The use of these fast transforms will benefit the Schur complement CG algorithm because $\mathbf{V}$ is used twice in each iteration. In the preconditioned LSQR algorithm, $\mathbf{V}$ only appears in the initialization and finalization stages.

For comparison we also created preconditioning subspaces from

- The SVD; the optimal subspace is created from the principal k right singular vectors of $\mathbf{K}$.
- regutm; the Regularization Tools package [7] includes a test problem generator regutm that produces random singular vectors with $j-1$ sign changes (where j is the column index).

5 Numerical Results.

The numerical tests were performed using MATLAB [10] with double precision IEEE arithmetic. The wavelet routines are from the wavelet package by Nielsen [11], and the discrete cosine transform is from MATLAB's Signal Processing Toolbox. The test problem heat is from Regularization Tools [7] and was selected because an unpreconditioned CG method shows slow convergence for this problem.

We also test the algorithms on an inverse geomagnetic problem originating from geoscience [1], where the goal is to compute the distribution of magnetic dipole moment from remote measurements. The relationship is described by a Fredholm integral equation of the first kind

$$\int_{\Omega} K(x, y, z, x', y', z')\, f(x, y, z)\, dx\, dy\, dz = g(x', y', z')$$

where the kernel K is given by

$$K(x, y, z, x', y', z') = \frac{3(z - z')}{r^5} - \frac{1}{r^3}$$

with $r = \big((x - x')^2 + (y - y')^2 + (z - z')^2\big)^{1/2}$.

The solution to the discretized problem consists of samples of f on a 3-D grid with $n = N^3$ points. The right-hand side consists of data measured at P layers, each layer consisting of N^2 data points and thus leading to a total of $m = PN^2$ data. The problem used here has parameters $N = 12$ and $P = 13$ leading to a matrix $\mathbf{K}$ of dimensions 1872×1728.

This formulation of the problem gives us the option to create a smaller underdetermined problem with $P = 1$, i.e., only one layer of data points, leading to a matrix of dimensions 144×1728. We use the right singular vectors of this matrix to create a preconditioning subspace for the overdetermined system.

More extensive numerical experiments with the Schur complement CG algorithm are available in [9], while experiments with our new method in geomagnetic problems can be found in [1].

5.1 Condition numbers.

Even though the condition number does not tell the entire truth about the convergence of CG and LSQR, it still carries information on what to expect. Furthermore it is independent of the right-hand side and we are able to draw conclusions on a more general level. The actual convergence depends on the distribution of *all* the singular values, i.e., not only the extreme singular values, as well as the actual right-hand side.

Figure 5.1 shows the condition number of the Schur complement $\mathbf{S}$ for the `heat` problem, as a function of the dimension k of the preconditioning subspace. The condition number using the wavelet basis decreases each time a detail level is "completed," while the other subspace types lead to smoother decay of the condition number. The basis derived from the SVD gives the smallest condition number, as expected.

5.2 Precision of subspace precondition LSQR vs. Schur CG.

It is known that LSQR often achieves higher precision than symmetric CG on the normal equations, even though the two methods are algebraically equivalent. Figure 5.2 shows the convergence histories and, as expected, we see that LSQR converges a bit faster than Schur CG. We emphasize that the LSQR algorithm is much faster, as the cost per iteration is about half that of Schur CG.

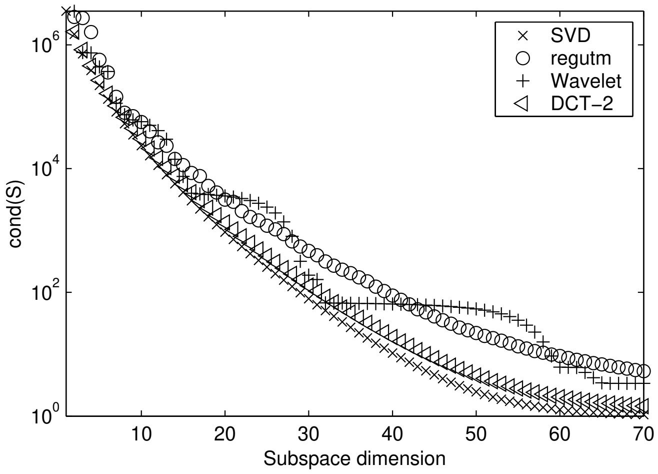


Figure 5.1: Condition number of the Schur complement $\mathbf{S}$ as a function of subspace dimension k. The test problem is heat with $m = n = 1024$, $\mathbf{L} = \mathbf{I}$ and $\lambda = 10^{-4}$.

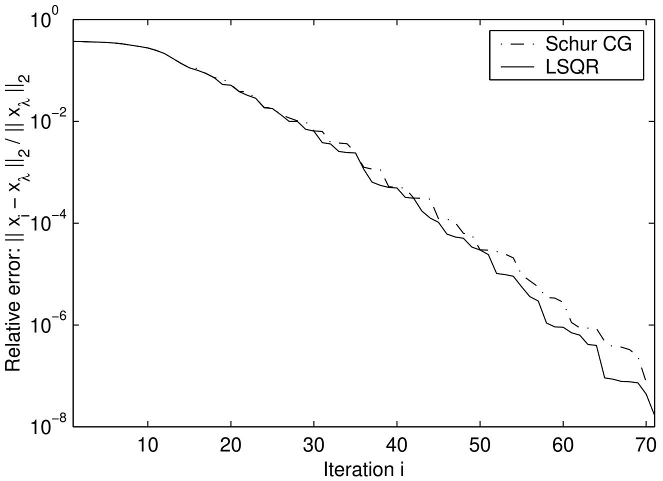


Figure 5.2: Schur complement CG and subspace preconditioned LSQR applied to heat with $m = n = 512$, $\mathbf{L} = \mathbf{I}$ and $\lambda = 10^{-4}$. The preconditioning subspace is a DCT-2 subspace of dimension $k = 20$.

5.3 Convergence with different subspace types.

Figure 5.3 shows convergence histories for the heat problem with fixed subspace dimensions 32. We see that the SVD-based preconditioner is fastest, followed by a DCT-2 based subspace. Compared to the unpreconditioned LSQR, convergence is very fast for this problem.

Turning to the geomagnetic problem we obtain the convergence history shown in Figure 5.4. Without the preconditioner convergence is almost non-existent, while LSQR with subspace preconditioning does converge. The DCT-2 subspace works nicely even though it does not take the structure of the problem into account. We also tried the SVD-based subspace from the small version of the problem, which yields even faster convergence.

5.4 Preconditioned LSQR on the unregularized problem.

Some iterative methods in their unpreconditioned version exhibit "semi-convergence" on unregularized ill-posed problems; i.e., during the initial iterations the solution approaches a regularized solution but at some point starts to converge towards the unwanted least squares solution, cf. §6 in [6]. This implies that early termination of the CG iterations is sometimes a regularization method in itself. CG and hence LSQR in their unpreconditioned versions are known to posses this semi-convergence property. Figure 5.5 illustrates that this is also the case for the preconditioned version of LSQR. Note that the smallest error of the preconditioned method is slightly larger than that of the plain (unpreconditioned) version, but it is reached in fewer iterations.

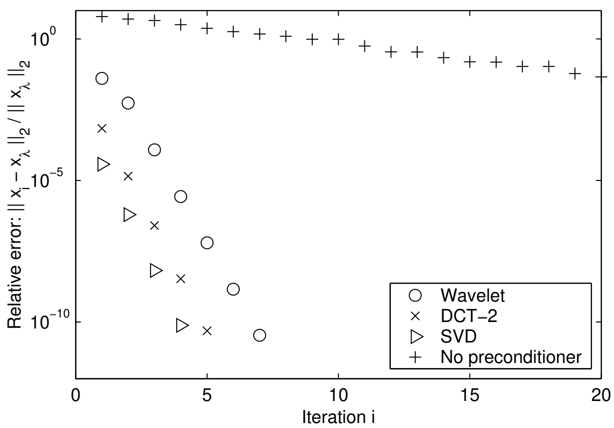


Figure 5.3: Relative error with respect to the *regularized solution* (not the true solution) as a function of iterations for three different subspace types with $k = 32$. The test problem is heat with $m = n = 1024$, $\mathbf{L} = \mathbf{I}$ and $\lambda = 10^{-5}$.

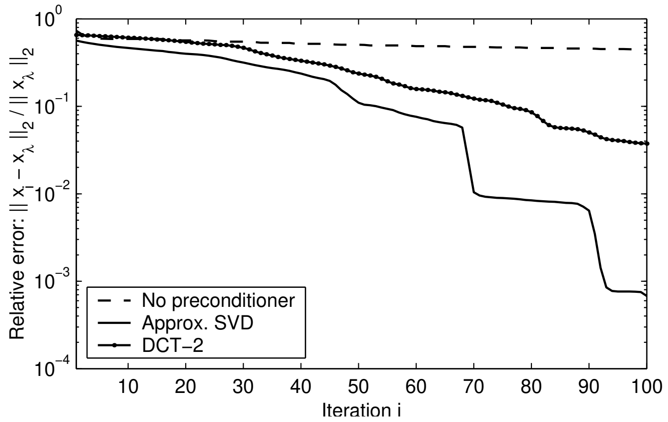


Figure 5.4: Subspace preconditioned LSQR on the geomagnetic problem. The matrix $\mathbf{K}$ is 1872×1728, i.e., 12^3 solution points and 13 layers of 12^2 measurements. The SVD-based preconditioning subspace consists of the right singular vectors from the SVD of a smaller 144×1728 single-layer problem. The regularization parameter is $\lambda = 10^{-2}$, and $\mathbf{L} = \mathbf{I}$.

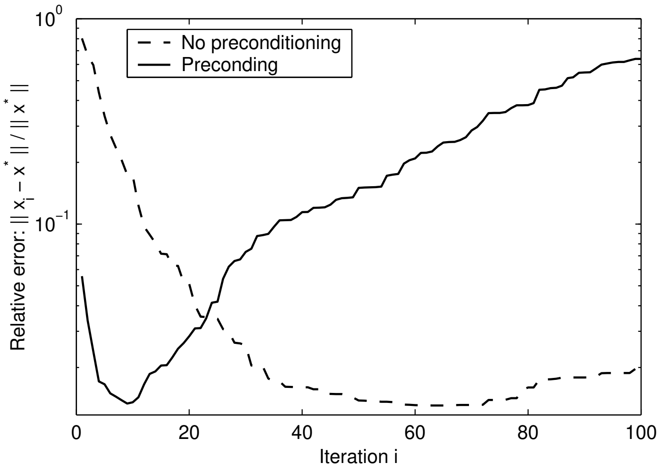


Figure 5.5: Plain and preconditioned LSQR applied to the unregularized problem heat with $m = n = 512$, $\mathbf{L} = \mathbf{0}$, noise level $\|\mathbf{e}\|_2 = 10^{-4}\|\mathbf{y}\|_2$, and a DCT-2 subspace of dimension $k = 20$. Both methods exhibit "semi-convergence."

Table 5.1: Timings on the heat problem with $m = n = 1024$, $\mathbf{L} = \mathbf{I}$, $\lambda = 10^{-5}$ and a DCT-2 basis of dimension $k = 8$. The iteration numbers are selected to give a relative error of 10^{-3} in the solution. All operations are dense, and the LSQR implementation is from Regularization Tools [7].

	LSQR	Schur CG with $\mathbf{g}^{(i)}$	Schur CG without $\mathbf{g}^{(i)}$	Precond. LSQR
Initialization (sec)	≈ 0	3	3	3
No. iterations	378	33	33	33
Iteration time (sec)	74	29	13	6
Finalization (sec)	0	0	0	≈ 0
Total time (sec)	74	32	16	9

5.5 Timings.

In the previous sections we only took the iteration numbers into account when evaluating the algorithms. However, the time per iteration is also vital. In addition, the initialization phase, i.e., the creation and factorization of $\widehat{\mathbf{K}}\mathbf{V}$, should be taken into account.

If the subspace transformations are implemented with matrix multiplications we see that the initialization phase costs at least k multiplications with $\widehat{\mathbf{K}}$ to construct $\widehat{\mathbf{K}}\mathbf{V}$. This corresponds to $k/2$ iterations with a standard LSQR or CGLS. Using more efficient subspace transformations like the DCT reduces this startup penalty and is to be recommended. However, if $\widehat{\mathbf{K}}$ is sparse the matrix multiplication approach could be the best choice. Furthermore the initialization phase includes a QR factorization of $\widehat{\mathbf{K}}\mathbf{V}$.

In situations where the same system is to be solved several times one can save and reuse the factorized $\widehat{\mathbf{K}}\mathbf{V}$, thus eliminating the initialization step.

Table 5.1 shows timings for LSQR without preconditioning, Schur complement CG and subspace preconditioned LSQR, broken into three phases. The problem size was selected so that everything was contained in memory. In the Schur complement CG algorithm one can skip step 11 and the variable $\mathbf{g}^{(i)}$ when $\mathbf{L} = \mathbf{I}$, which saves substantial time in our MATLAB implementation.

We see that the time for the initialization phase is equal for the two implementations of the two-level methods—essentially they both require the creation of $\widehat{\mathbf{K}}\mathbf{V}$ and a factorization of this matrix. The finalization stages are negligible.

In the iteration phase we see the strength of the new implementation; a preconditioned LSQR iteration is between two to four times faster than a Schur CG iteration, depending on the matrix $\mathbf{L}$ and the implementation of step 11. Both implementations require 33 iterations to achieve the desired accuracy, while plain LSQR requires 378 iterations.

Note that for preconditioned LSQR, the algorithm of section 3.2 was used throughout. The same graphs would be obtained with the section 3.3 modifications, but the operations with $\mathbf{W}$ would increase the timings slightly.

6 Conclusion.

Least squares problems may be preconditioned on the left by any orthogonal matrix, and on the right by any nonsingular matrix. Here we are using $\mathbf{Q}^{\mathrm{T}}$ and $[\,\mathbf{V}\;\mathbf{W}\,]$ respectively to decouple the problem into two parts: (3.2) and (3.3). We have shown that the resulting preconditioned LSQR method is equivalent to Schur complement CG. Both methods are adapted to avoid operations with $\mathbf{W}$, but the implementation is simpler for subspace preconditioned LSQR and the work per iteration is halved.

The subspace $\mathbf{V}$ must be chosen wisely for the preconditioning to give fast convergence. The DCT-2 subspace produced good results, while the wavelet subspace turned out to be weaker for the problems we considered. We have illustrated that in some cases a good subspace can be derived from a simpler problem. Because the initialization phase can be expensive in terms of operations, the subspace dimension should be chosen small.

We also observed that the Schur complement CG and subspace preconditioned LSQR algorithms both exhibit semi-convergence on unregularized problems. Thus, regularization can be achieved by early termination of the iterations.

Acknowledgements.

We are grateful to the referees for finding some important corrections. We also acknowledge the SIAM annual meeting (San Diego, 2001) and the IMM Sparse Matrix Computations summer school (Danish Technical University, 2001) for helping this collaboration evolve.

REFERENCES

1. M. Andersen, M. Fedi, P. Hansen, and V. Paoletti, *SVD and GSVD analysis of depth resolution in potential field inversion*, Inverse Problems, (Submitted).
2. O. Axelsson, *Iterative Solution Methods*, Cambridge University Press, 1994.
3. R. Barret, M. Berry, T. F. Chan, J. Demmel, J. Danato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994.
4. I. Daubechies, *Ten Lectures on Wavelets*, SIAM, 1992.
5. M. Hanke and C. R. Vogel, *Two-level preconditioners for regularized inverse problems I: Theory*, Numerische Mathematik, 83 (1999), pp. 385–402.
6. P. C. Hansen, *Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion*, SIAM, 1998.
7. ——, *Regularization tools version 3.0 for Matlab 5.2*, Numerical Algorithms, 20 (1999), pp. 195–196. Download from `http://www.imm.dtu.dk/~pch/Regutools/regutools.html`.

8. M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards, 49 (1952), pp. 409–436.
9. M. Jacobsen, *Two-grid iterative methods for ill-posed problems*, Master's thesis, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark, September 2000. Available throug `http://www.imm.dtu.dk/~mj/`.
10. The MathWorks, Inc., *Matlab Manual*, Natick, MA, 6.0 ed., 2000.
11. O. M. Nielsen, *Wavelets in Scientific Computing*, PhD thesis, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark, July 1998. Available throug `http://www.imm.dtu.dk/~omni`.
12. C. C. Paige and M. A. Saunders, *LSQR: An algorithm for sparse linear equations and sparse least squares*, ACM Transactions on Mathematical Software, 8 (1982), pp. 43–71.
13. ——, *LSQR: Sparse linear equations and least squares problems*, ACM Transactions on Mathematical Software, 8 (1982), pp. 195–209.
14. K. L. Riley, *Two-Level Preconditioners for Regularized Ill-Posed Problems*, PhD thesis, Montana State University – Bozeman, 1999.
15. K. L. Riley and C. R. Vogel, *Two-level preconditioners for ill-conditioned linear systems with semidefinite regularization*. Submitted to Elsevier.
16. G. Strang, *The discrete cosine transform*, SIAM Review, 41 (1999), pp. 135–147.
17. C. R. Vogel, *Negative results for multilevel preconditioners in image deblurring*, in Scale-Space Theories in Computer Vision, M. Nielsen, P. Johansen, O. F. Olsen, and J. Weickert, eds., Springer, 1999, pp. 489–494.
18. C. R. Vogel, *Computational Methods for Inverse Problems*, SIAM, to appear.

Bibliography

[1] D. Adams, *The Ultimate Hitchhiker's Guide - Complete and Unabridged*, Random House, New York, NY, 1996, ch. The Hithchiker's Guide to the Galaxy.

[2] H.-M. Adorf, *Hubble Space Telescope image restoration in its fourth year*, Inverse Problems, 11 (1995), pp. 639–653.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, SIAM, 3 ed., 1999.

[4] C. T. H. Baker, *The Numerical Treatment of Integral Equations*, Oxford University Press, 1977.

[5] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 1993.

[6] V. Baryamureeba, *The impact of equal weighting of low- and high-confidence observations on robust linear regression computations*, BIT, 41 (2001), pp. 847–855.

[7] M. Belge, M. E. Kilmer, and E. L. Miller, *Efficient determination of multiple regularization parameters in a generalized L-curve framework*, Inverse Problems, 18 (2002), pp. 1161–1183.

[8] A. Ben-Israel and T. N. E. Greville, *Generalized Inverses: Theory and Applications*, Robert E. Krieger Publishing, Huntington, NY, reprint ed., 1980.

[9] F. Berntsson and L. Eldén, *Numerical solution of a Cauchy problem for the Laplace equation*, Inverse Problems, 17 (2001).

[10] M. Bertero and P. Boccacci, *Introduction to Inverse Problems in Imaging*, Institute of Physics Publishing, Bristoll, 1998.

[11] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA BEGIN*, Auerbach Publishers Inc./Studentlitteratur, 1973.

[12] Å. Björck, *A bidiagonalization algorithm for solving large and sparse ill-posed systems of linear equations*, BIT, 28 (1988), pp. 659–670.

[13] ——, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, PA, 1996.

[14] Å. Björck, E. Grimme, and P. van Dooren, *An implicit shift bidiagonalization algorithm for ill-posed systems*, BIT, 34 (1994), pp. 510–534.

[15] E. Y. Bobrovnikova and S. A. Vavasis, *Accurate solution of weighted least squares by iterative methods*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 1153–1174.

[16] C. A. Brebbia and J. Dominguez, *Boundary Elements: An Introductory Course*, McGraw-Hill Inc., 1989.

[17] C. Brezinski, M. Redivo-Zaglia, G. Rodriguez, and S. Seatzu, *Multiparameter regularization techniques for ill-conditioned linear systems*, Numerische Mathematik, 94 (2003), pp. 203–228.

[18] D. Calvetti, P. C. Hansen, and L. Reichel, *L-curve curvature bounds via Lanczos bidiagonalization*, Electronic Transactions on Numerical Analysis, 14 (2002), pp. 134–147.

[19] D. Calvetti, B. Lewis, and L. Reichel, *On the regularizing properties of the GMRES method*, Numerische Mathematik, 91 (2002).

[20] D. Calvetti, L. Reichel, and A. Shuibi, *Enriched Krylov subspace methods for ill-posed problems*, Linear Algebra and its Applications, 362 (2003), pp. 257–273.

[21] D. Cann, *Retire Fortran? A debate rekindled*, Communications of the ACM, 35 (1992), pp. 81–89.

[22] R. H. Chan, J. G. Nagy, and R. J. Plemmons, *FFT-based preconditioners for Toeplitz-block least squares problems*, SIAM J. Numer. Anal., 30 (1993), pp. 1740–1768.

[23] ——, *Circulant preconditioned Toeplitz least squares iterations*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 80–97.

[24] R. H. Chan and M. K. Ng, *Conjugate gradient methods for toeplitz systems*, SIAM Review, (1996).

[25] J. F. Claerbout and F. Muir, *Robust modeling with erratic data*, Geophysics, 38 (1973), pp. 826–844.

[26] D. Coleman, P. Holland, N. Kaden, V. Klema, and S. C. Peters, *A system of subroutines for iteratively reweighted least squares computations*, ACM Transactions on Mathematical Software, 6 (1980), pp. 327–336.

[27] COMSOL AB, *FEMLAB*, Stockholm, Sweden.

[28] P. J. Davis and P. Rabinowitz, *Methods of Numerical Integration*, Computer Science and Applied Mathematics, Academic Press, Orlando, FL, second ed., 1984.

[29] G. D. de Villiers, B. McNally, and E. R. Pike, *Positive solutions to linear inverse problems*, Inverse Problems, 15 (1999), pp. 615–635.

[30] L. M. Delves and J. Walsh, eds., *Numerical Solution of Integral Equations*, Oxford University Press, 1974.

[31] E. J. Dudewicz and S. N. Mishra, *Modern Mathematical Statistics*, Probability and Mathematical Statistics, John Wiley & Sons, New York, 1988.

[32] H. Ekblom, *l_p-methods for robust regression*, BIT, 14 (1974), pp. 22–32.

[33] L. Eldén, *Algorithms for the regularization of ill-conditioned least squares problems*, BIT, 17 (1977), pp. 134–145.

[34] ——, *A program for interactive regularization: Part I: Numerical algorithms*, Tech. Rep. LiTH-MAT-R-79-25, Dept. of Mathematics, Linköping Univ., August 1979.

[35] ——, *A weighted pseudoinverse, generalized singular values, and constrained least squares problems*, BIT, 22 (1982), pp. 487–502.

[36] H. W. Engl, M. Hanke, and A. Neubauer, *Regularization of Inverse Problems*, Mathematics and Its Applications, Kluwer Academic Publishers, 1996.

[37] H. W. Engl, A. K. Louis, and W. Rundell, eds., *Inverse Problems in Geophysical Applications*, Philadelphia, PA, 1997, SIAM.

[38] K. Engø, A. Marthinsen, and H. Z. Munthe-Kaas, *Diffman user's guide, version 1.6*, Tech. Rep. 166, Department of Informatics, University of Bergen, Norway, 1999.

[39] R. Fletcher, *Practical Methods of Optimization*, John Wiley & Sons, second ed., 1987.

[40] P. E. Frandsen, K. Jonasson, H. B. Nielsen, and O. Tingleff, *Unconstrained optimization*, Lecture Note 2, IMM, DTU, 1999.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, Addison-Wesley, 1994.

[42] D. A. Girard, *A fast 'Monte-Carlo cross-validation' procedure for large least squares problems with noisy data*, Numerische Mathematik, 56 (1989), pp. 1–23.

[43] ——, *Asymptotic optimality of fast randomized versions of GCV and c_l in ridge regression and regularization*, Ann. Statist., 19 (1991), pp. 1950–1963.

[44] G. H. Golub, *Some modified matrix eigenvalue problems*, SIAM Review, 15 (1973), pp. 318–334.

[45] G. H. Golub and W. Kahan, *Calculating the singular values and pseudoinverse of a matrix*, SIAM J. Numer. Anal., 2 (1965), pp. 205–224.

[46] G. H. Golub and C. F. V. Loan, *Matrix Computations*, John Hopkins University Press, third ed., 1996.

[47] G. H. Golub and G. Meurant, *Matrices, moments and quadrature*, in Numerical Analysis 1993, D. F. Griffith and G. A. Watson, eds., Ptman Research Notes in Mathematical Sciences, Pitman, New York, 1994, pp. 105–156.

[48] G. H. Golub and U. von Matt, *Quadratically constrained least squares and quadratic problems*, Numer. Math., 59 (1991), pp. 561–580.

[49] ——, *Generalized cross-validation for large-scale problems*, Journal of Computational and Graphical Statistics, 6 (1997), pp. 1–34.

[50] ——, *Tikhonov regularization for large scale problems*, in Scientific Computing, G. H. Golub, ed., Springer, 1997, pp. 3–26.

[51] R. Gonin and A. H. Money, *Nonlinear L_p-norm Estimation*, vol. 100 of Statistics: Textbooks and Monographs, Marcel Dekker, Inc., New York, 1989.

[52] J. Gosling, B. Joy, and G. S. G. Bracha, *The Java Language Specification*, The Java Series, Addison-Wesley, Reading, MA, second ed., 2000.

[53] C. W. Groetsch, *Inverse Problems in the Mathematical Sciences*, Mathematics for Scientists and Engineers, Vieweg, Braunschweig/Wiesbaden, 1993.

[54] ——, *Inverse Problems: Activities for Undergraduates*, Mathematical Associations of America, 1999.

[55] M. Gulliksson and P.-A. Wedin, *Optimization tools for Tikhonov regularization of nonlinear equations using the L-curve and its dual*, in Hansen et al. [69].

[56] J. Hadamard, *Lectures on Cauchy's Problem in Linear Partial Differential Equations*, Yale University Press, New Haven, CT, 1923.

[57] M. Hanke, *Conjugate Gradient Type Methods for Ill-Posed Problems*, Pitman Research Notes in Mathematics, Longman House, 1995.

[58] ——, *A note on Tikhonov regularization of large linear problems*, BIT, (2003), pp. 449–451.

[59] M. Hanke and P. C. Hansen, *Regularization methods for large-scale problems*, Surveys on Mathematics for Industry, 3 (1993), pp. 253–315.

[60] M. Hanke and J. G. Nagy, *Restoration of atmospherically blurred images by symmetric indefinite conjugate gradient techniques*, Inverse Problems, 12 (1996), pp. 157–173.

[61] M. Hanke, J. G. Nagy, and R. Plemmons, *Preconditioned iterative regularization for ill-posed problems*, in Numerical Linear Algebra, L. Reichel, A. Ruttan, and R. S. Varga, eds., Walter de Gruyter & Co., Berlin, 1993, pp. 141–163.

[62] M. Hanke and C. Vogel, *Two-level preconditioners for regularized inverse problems I: Theory*, Numerische Mathematik, 83 (1999).

[63] P. C. Hansen, *Regularization, GSVD and truncated GSVD*, BIT, 29 (1989), pp. 491–504.

[64] ——, *The discrete Picard condition for discrete ill-posed problems*, BIT, (1990).

[65] ——, *Analysis of discrete ill-posed problems by means of the L-curve*, SIAM Review, 34 (1992), pp. 561–580.

[66] ——, *Rank-deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion*, SIAM Monographs on Mathematical Modeling and Computation, SIAM, Philadelphia, PA, 1998.

[67] ——, *Regularization tools: A Matlab package for analysis and solution of discrete ill-posed problems, version 3.1 for Matlab 6.0*, Tech. Rep. IMM-TR-1998-6, Informatics and Mathematical Modelling, Technical University of Denmark, 1998.

[68] ——, *Regularization tools version 3.0 for Matlab 5.2*, Numerical Algorithms, 20 (1999), pp. 195–196.

[69] P. C. Hansen, B. H. Jacobsen, and K. Mosegaard, eds., *Methods and Applications of Inversion*, Lecture Notes in Earth Sciences, Springer, 2000.

[70] P. C. Hansen, M. Jacobsen, J. M. Rasmussen, and H. Sørensen, *The PP-TSVD algorithm for image restoration problems*, in Hansen et al. [69], pp. 171–186.

[71] P. C. Hansen, M. Jacobsen, and M. A. Saunders, *Subspace preconditioned LSQR for discrete ill-posed problems*, BIT, 43 (2003), pp. 975–989.

[72] P. C. Hansen and D. P. O'Leary, *The use of the L-curve in the regularization of discrete ill-posed problems*, SIAM J. Sci. Comput, 14 (1993), pp. 1487–1503.

[73] P. C. Hansen, T. Sekii, and H. Shibahashi, *The modified truncated SVD method for regularization in general form*, SIAM J. Sci. Stat. Comput., 13 (1992), pp. 1142–1150.

[74] B. Hayes, *A lucid interval*, American Scientist, (2003).

[75] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards, 49 (1952), pp. 409–436.

[76] J. Hicklin, C. Moler, P. Web, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington, *JAMA: A Java Matrix Package*, The Mathworks and NIST. See `http://math.nist.gov/javanumerics/jama/`.

[77] R. A. Horn and C. R. Johnson, *Topics in Matrix Analysis*, Cambridge University Press, 1991.

[78] P. J. Huber, *Robust Estimation*, John Wiley & Sons, 1981.

[79] J. Hughes, *Why functional programming matters*, The Computer Journal, 32 (1989), pp. 98–107.

[80] M. F. Hutchinson, *A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines*, Communication in Statistics, Simulation and Computation, 19 (1990), pp. 433–450.

[81] M. Jacobsen, *Two-grid iterative methods for ill-posed problems*, Master's thesis, Informatics and Mathematical Modelling, Techinical University of Denmark, 2000. Available from `http://www.imm.dtu.dk/documents/ftp/ep2000/ep27_00.abstract.html`.

[82] T. K. Jensen, *Regularizing iterations for image restoration*, Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, 2003.

[83] S. P. Jones, ed., *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, Cambridge, 2003. Special issue of "Journal of Functional Programming".

[84] J. Kamm and J. G. Nagy, *Kronecker product and SVD approximations in image restoration*, Linear Algebra and its Applications, 284 (1998), pp. 177–192.

[85] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, second ed., 1988.

[86] M. E. Kilmer, *Cauchy-like preconditioners for two-dimensional ill-posed problems*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 777–799.

[87] M. E. Kilmer and D. P. O'Leary, *Choosing regularization parameters in iterative methods for ill-posed problems*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 1204–1221.

[88] M. E. Kilmer and G. W. Stewart, *Iterative regularization and MINRES*, SIAM J. Matrix Anal. Appl., 21 (1999), pp. 613–628.

[89] H. P. Langtangen, *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*, Lecture Notes in Computational Science and Engineering, Springer, 1999.

[90] C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[91] Y. Li, *A globally convergent method for l_p problems*, SIAM J. Optimization, 3 (1993), pp. 609–629.

[92] C. V. Loan, *Computational Frameworks for the Fast Fourier Transform*, Frontiers in Applied Mathematics, SIAM, Philadelphia, PA, 1992.

[93] K. Madsen and H. B. Nielsen, *A finite smoothing algorithm for linear l_1 estimation*, SIAM J. Optim., 3 (1993), pp. 223–235.

[94] The MathWorks Inc., *Using MATLAB Version 6*, Natick, MA, 2002.

[95] S. Mehrotra, *On the implementation of a primal-dual interior point method*, SIAM J. Optim., (1992).

[96] M. Metcalf and J. Reid, *Fortran 90/95 Explained*, Oxford University Press, Oxford, 1996.

[97] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML (Revised)*, MIT Press, Cambridge, MA, 1997.

[98] V. A. Morozov, *Methods for Solving Incorrectly Posed Problems*, Springer-Verlag, 1984. Translated from Russian.

[99] ——, *Regularization Methods for Ill-Posed Problems*, CRC Press, Boca Ranton, FL, 1993. Translated from Russian.

[100] J. G. Nagy, L. Perrone, and K. P. Lee, *Iterative methods for image restoration: A Matlab object oriented approach.* Available from `http://www.mathcs.emory.edu/~nagy/RestoreTools/index.html`.

[101] M. Z. Nashed, ed., *Generalized Inverses and Applications*, Academic Press, 1976.

[102] F. Natterer, *The Mathematics of Computerized Tomography*, John Wiley & Sons, 1986.

[103] D. P. O'Leary, *Robust regression computation using iteratively reweighted least squares*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 466–480.

[104] D. P. O'Leary and J. A. Simmons, *A bidiagonalization-regularization procedure for large scale discretizations of ill-posed problems*, SIAM J. Sci. Stat. Comput., 2 (1981), pp. 474–488.

[105] C. C. Paige and M. A. Saunders, *Solution of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal., 12 (1975), pp. 617–629.

[106] ——, *Algorithm 583: LSQR: Sparse linear equations and least squares problems*, ACM Transactions on Mathematical Software, 8 (1982), pp. 195–209.

[107] ——, *LSQR: An algorithm for sparse linear equations and sparse least squares*, ACM Transactions on Mathematical Software, 8 (1982), pp. 43–71.

[108] L. C. Paulsen, *ML for the Working Programmer*, Cambridge University Press, 1991.

[109] J. M. Rasmussen, *Compact linear operators and Krylov subspace methods*, Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, 2001.

[110] T. Regińska, *A regularization parameter in discrete ill-posed problems*, SIAM J. Sci. Comput., 17 (1996), pp. 740–749.

[111] D. Rémy and J. Vouillon, *Objective ML: An effective object-oriented extension to ML*, Theory and Practice of Object Systems, 4 (1998), pp. 27–50.

[112] M. Rojas, S. A. Santos, and D. C. Sorensen, *A new matrix-free algorithm for the large-scale trust-region problem*, SIAM J. Optim, 11 (2000).

[113] M. Rojas and D. C. Sorensen, *A trust-region approach to the regularization of large-scale discrete forms of ill-posed problems*, SIAM J. Sci. Comput., 23 (2002), pp. 1843–1961.

[114] M. Rojas and T. Steihaug, *An interior-point trust-region-based method for large-scale non-negative regularization*, Inverse Problems, 18 (2002).

[115] R. Rucker, *Software Engineering and Computer Games*, Addison Wesley, 2003.

[116] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856–869.

[117] A. Schuhmacher, J. Hald, K. B. Rasmussen, and P. C. Hansen, *Sound source reconstruction using inverse boundary element calculations*, J. Acoust. Soc. Am., 113 (2003), pp. 114–127.

[118] G. L. Steele Jr., *Common LISP, The Language*, Digital Press, Bedford, MA, second ed., 1990.

[119] T. Steihaug, *The conjugate gradient method and trust regions in large scale optimization*, SIAM J. Num. Anal., 20 (1983), pp. 626–637.

[120] I. Stewart, *Never ending story*, New Scientist, 179 (2003), pp. 28–33.

[121] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, third ed., 1997.

[122] A. Tarantola, *Inverse Problem Theory: Methods for Data Fitting and Model Parameter Estimation*, Elsevier, Amsterdam, 1987.

[123] A. N. Tikhonov and V. Y. Arsenin, *Solutions of Ill-Posed Problems*, Scripta Series in Mathematics, John Wiley & Sons, New York, 1977.

[124] C. R. Vogel, *Computational Methods for Inverse Problems*, Frontiers in Applied Mathematics, SIAM, 2002.

[125] G. Wahba, *Ill-posed problems: Numerical and statistical methods for mildly, moderately and severely ill-posed problems with noisy data*, SIAM J. Numer. Anal., 14 (1977), pp. 651–667.

[126] G. M. Wing and J. D. Zahrt, *A Primer on Integral Equations of the First Kind: The Problem of Deconvolution and Unfolding*, SIAM, 1991.

[127] N. Wirth, *The programming language PASCAL*, Acta Informatica, 1 (1971), pp. 35–63.

[128] S. J. Wright, *Primal-Dual Interior-Point Methods*, SIAM, 1997.