Algorithms for Cholesky and QR Factorizations, and the Semidefinite Generalized Eigenvalue Problem

Lucas, Craig

2004

MIMS EPrint: **2007.5**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

ALGORITHMS FOR CHOLESKY AND QR FACTORIZATIONS, AND THE SEMIDEFINITE GENERALIZED EIGENVALUE PROBLEM

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

2004

Craig Lucas
Department of Mathematics

Contents

List of Tables

List of Figures

Abstract

We consider algorithms for three problems in numerical linear algebra: computing the pivoted Cholesky factorization, solving the semidefinite generalized eigenvalue problem and updating the QR factorization.

Fortran 77 codes exist in LAPACK for computing the Cholesky factorization (without pivoting) of a symmetric positive definite matrix using Level 2 and 3 BLAS. In LINPACK there is a Level 1 BLAS routine for computing the Cholesky factorization with complete pivoting of a symmetric positive *semi*definite matrix. We present two new algorithms and Fortran 77 LAPACK-style codes for computing this pivoted factorization: one using Level 2 BLAS and one using Level 3 BLAS. We show that on modern machines the new codes can be many times faster than the LINPACK code. Also, with a new stopping criterion they provide more reliable rank detection and can have a smaller normwise backward error.

The generalized eigenvalue problem $Ax = \lambda Bx$ in the case where A and B are real and symmetric and B is positive semidefinite is considered. We present an algorithm for solving this problem that has a potentially smaller operation count than existing methods and requires no further restrictions on A and B. The eigenvalues of the problem are classified as finite or infinite. Nonregular matrix pencils, where the eigenvalues can take any value, are also discussed and

a deflation strategy is given. We include a MATLAB code for our algorithm and give some numerical experiments.

We also treat the problem of updating the QR factorization, with applications to the least squares problem. Algorithms are presented that compute the factorization $\widetilde{A} = \widetilde{Q}\widetilde{R}$ where $\widetilde{A}$ is the matrix $A = QR$ after it has had a number of rows or columns added or deleted. This is achieved by updating the factors Q and R, and we show this can be much faster than computing the factorization of $\widetilde{A}$ from scratch. We consider algorithms that exploit the Level 3 BLAS where possible and place no restriction on the dimensions of A or the number of rows and columns added or deleted. For some of our algorithms we present Fortran 77 LAPACK-style code and show the backward error of our updated factors is comparable to the error bounds of the QR factorization of $\widetilde{A}$.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

The material in Chapter 2 and Appendix A is based on the technical report 'LAPACK-Style Codes for Level 2 and 3 Pivoted Cholesky Factorizations', Numerical Analysis Report No. 442, Manchester Centre for Computational Mathematics, February, 2004 (`http://www.ma.man.ac.uk/MCCM/`). The report is also LAPACK Working Note 161 (`http://www.netlib.org/lapack/lawns/`).

Copyright

Acknowledgments

The following people all contributed to the writing of this thesis. I am extremely grateful to all of them.

My supervisor Nick Higham shared his expertise and guided me through the writing process, without him it would not have been possible.

Sven Hammarling gave me much advice and support. I had many useful exchanges with him and he taught me a lot about scientific programming.

Françoise Tisseur made many helpful suggestions throughout my three years as a PhD student.

Volker Mehrmann made useful comments on the material in Chapter 3.

The company of my fellow students Michael Berhanu, Gareth Hargreaves and Anna Mills was both helpful to my work and a welcome distraction from it.

Chris Paul supplied fast and detailed IT support when I needed it.

Finally, Glenn whose constant encouragement was integral to the completion of this work. His love, affection and humour enabled me to switch off and relax when I needed to. This thesis is dedicated to him.

Chapter 1

Introduction

We start by giving some definitions and background theory which we use throughout this thesis. Some of the material is this section is based on material in [14], [24], [29], [30], [41], [50] and [51].

1.1 Basic Definitions

1.1.1 Definitions in Linear Algebra

- A set of vectors $S = \{a_1, \ldots, a_n\} \in \mathbb{R}^m$ is *linearly independent* if the *linear combination*

$$\sum_{i=1}^{n} \alpha_i a_i, \quad \alpha_i \in \mathbb{R}$$

being zero implies all the α_i are zero. Otherwise, if a nontrivial ($\alpha_i \neq 0, i = 1\!:\!n$) combination of the a_i is zero then the vectors in S are said to be *linearly dependent*.

- The set of all linear combinations of S is called the *span* of S, that is

$$\text{span}\{a_1, \ldots, a_n\} = \Big\{ \sum_{i=1}^{n} \beta_i a_i : \beta_i \in \mathbb{R} \Big\}.$$

If the vectors in S are linearly independent and $b \in \text{span}(S)$, then b is a unique linear combination of the a_i.

- The *range* of a matrix A is defined by

$$\text{ran}(A) = \{y \in \mathbb{R}^m : y = Ax \text{ for some } x \in \mathbb{R}^n\}.$$

- The *null space* of A is defined by

$$\text{null}(A) = \{x \in \mathbb{R}^m : Ax = 0\}.$$

- If the matrix $A = [a_1 \dots a_n]$ is a column partitioning then

$$\text{ran}(A) = \text{span}\{a_1, \dots, a_n\}.$$

- The *rank* of a A is defined by

$$\text{rank}(A) = \dim(\text{ran}(A)),$$

where dim is the *dimension*, the number of linearly independent vectors, of ran(A).

- If for $A \in \mathbb{R}^{m \times n}$

$$\text{rank}(A) < \min(m, n),$$

then A is *rank deficient*.

1.1.2 Matrix Types

- The matrix $A^T \in \mathbb{R}^{n \times n}$ is the *transpose* of A. If the (i, j) element of A is a_{ij}, then the (i, j) element of A^T is a_{ji}.

- The matrix $A^* \in \mathbb{C}^{n \times n}$ is the *conjugate transpose* of A. If the (i, j) element of A is a_{ij}, then the (i, j) element of A^* is $\bar{a}_{ji}$, where $\bar{a}_{ji}$ is the conjugate of a_{ji}.

- A matrix $A \in \mathbb{R}^{n\times n}$ is *symmetric* if $A = A^T$ and $A \in \mathbb{C}^{n\times n}$ is *Hermitian* if $A = A^*$.

- The *identity matrix* $I_n \in \mathbb{R}^{n\times n}$ is a matrix which has ones on the diagonal and zeros elsewhere. The vector e_k is the kth column of I_n.

- A matrix $Q \in \mathbb{R}^{n\times n}$ is *orthogonal* if $Q^TQ = I$, where I is the identity matrix and $Q \in \mathbb{C}^{n\times n}$ is *unitary* if $Q^*Q = I$.

- A matrix $A \in \mathbb{C}^{n\times n}$ is *upper triangular* if $a_{ij} = 0,\ i > j$. Similarly if $a_{ij} = 0,\ j > i$ then the matrix $A \in \mathbb{C}^{n\times n}$ is *lower triangular*. If the diagonal elements of an upper (or lower) triangular matrix are all 1, then the matrix is said to be *unit upper (or lower) triangular*.

- A matrix $A \in \mathbb{C}^{m\times n}$, $m \neq n$, is *upper trapezoidal* if $a_{ij} = 0,\ i > j$.

- A matrix $A \in \mathbb{C}^{m\times n}$ is *upper Hessenberg* if $a_{ij} = 0,\ i > j + 1$.

- A *permutation matrix* is a permutation of the identity matrix, where either the rows or columns are a permutation of those in I_n. The effect of multiplying a matrix from the left by a permutation matrix is the reordering of the rows of that matrix. Multiplying from the right reorders columns.

- If $AX = XA = I_n$ for $A, X \in \mathbb{R}^{n\times n}$ then X is said to be the *inverse* of A and is denoted A^{-1}. If A^{-1} exists then $\text{rank}(A) = n$ and it is said to have *full rank* and be *nonsingular*, else it is *singular*.

- A *block matrix* is a matrix regarded as consisting not of individual elements but of blocks of elements. Many of the above definitions extend to block matrices.

1.1.3 The Determinant

The *determinant* of an n-by-n matrix $A \in \mathbb{R}^{n \times n}$ is defined in terms of order $n-1$ determinants by

$$\det(A) = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} \det(A_{1j}),$$

where A_{1j} is an $(n-1)$-by-$(n-1)$ matrix obtained by deleting the first row and jth column of A. In the 1-by-1 case

$$\det(a) = a.$$

If the determinant of A is nonzero then this implies A is nonsingular and vice versa.

The determinant of an upper (or lower) triangular matrix is the product of the diagonal elements. The determinant of a block upper (or lower) triangular matrix is equal to the product of the determinants of the individual diagonal blocks.

1.1.4 Vector Norms

Norms are a means of obtaining a scalar measure of the size of a vector or matrix.

Vector norms are functions $\|\cdot\| : \mathbb{C}^n \to \mathbb{R}$ and satisfy the vector norm axioms:

- $\|x\| \geq 0$ for all $x \in \mathbb{C}^n$, and $\|x\| = 0$ if and only if $x = 0$.
- $\|\kappa x\| = |\kappa| \|x\|$ where $\kappa \in \mathbb{C}$, $x \in \mathbb{C}^n$.
- $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in C^n$ (triangle inequality).

The Hölder p-norm is defined as

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}, \quad p \geq 1.$$

This definition gives the three most popular vector norms:

$$\begin{aligned} \|x\|_1 &= \sum_{i=1}^{n} |x_i|, \\ \|x\|_2 &= \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2} = (x^* x)^{1/2}, \quad \text{(Euclidean Length)}, \\ \|x\|_\infty &= \max_{1 \leq i \leq n} |x_i|. \end{aligned}$$

The vector 2-norm is invariant under orthogonal or unitary transformation, that is

$$\|Qx\|_2^2 = x^* Q^* Q x = x^* x = \|x\|_2^2,$$

for orthogonal or unitary matrix Q.

1.1.5 Matrix Norms

Matrix norms are functions $\|\cdot\| : \mathbb{C}^{m \times n} \to \mathbb{R}$ and satisfy the matrix norm axioms:

- $\|A\| \geq 0$ for all $A \in \mathbb{C}^{m \times n}$, and $\|A\| = 0$ if and only if $A = 0$.
- $\|\kappa A\| = |\kappa| \|A\|$ where $\kappa \in \mathbb{C},\ A \in \mathbb{C}^{m \times n}$.
- $\|A + B\| \leq \|A\| + \|B\|$ for all $A, B \in \mathbb{C}^{m \times n}$.

Common matrix norms for $A \in \mathbb{C}^{m \times n}$ are the Frobenius norm, defined as

$$\|A\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2},$$

and the *subordinate* matrix norms defined as

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|},$$

for given vector norms on $\mathbb{C}^m$ in the numerator and $\mathbb{C}^n$ in the denominator.

Three useful subordinate matrix norms are:

$$\begin{aligned}
\|A\|_1 &= \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| \quad \text{(maximum column sum)},\\
\|A\|_2 &= (\rho(A^*A))^{1/2} = \sigma_{\max}(A), \quad \text{(spectral norm)},\\
\|A\|_\infty &= \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}| \quad \text{(maximum row sum)},
\end{aligned}$$

where the spectral radius

$$\rho(A) = \max\{|\lambda| : \det(A - \lambda I) = 0\},$$

and $\sigma_{\max}(A)$ is the largest singular value of A, see Section 1.5.4.

A matrix norm is *consistent* if

$$\|AB\| \le \|A\|\|B\|.$$

All the above matrix norms are consistent.

The Frobenius and 2-norm are invariant under orthogonal and unitary transformation, that is, for orthogonal or unitary matrices U and V,

$$\|UAV\|_2 = \|A\|_2 \quad \text{and} \quad \|UAV\|_F = \|A\|_F.$$

The definition of subordinate matrix norms can be generalized by allowing different vector norms, so we have

$$\|A\|_{\alpha,\beta} = \max_{x\neq 0} \frac{\|Ax\|_\beta}{\|x\|_\alpha}.$$

The matrix *condition number*, $\kappa_{\alpha,\beta}(A)$, of a nonsingular matrix A is defined in terms of these generalized subordinate norms

$$\kappa_{\alpha,\beta}(A) := \lim_{\epsilon\to 0} \sup_{\|\Delta A\|_{\alpha,\beta}\le\epsilon\|\Delta A\|_{\alpha,\beta}} \left(\frac{\|(A+\Delta A)^{-1} - A^{-1}\|_{\beta,\alpha}}{\epsilon\|A^{-1}\|_{\beta,\alpha}} \right).$$

And by [29, Thm. 6.4], equivalently

$$\kappa_{\alpha,\beta}(A) = \|A\|_{\alpha,\beta}\|A^{-1}\|_{\beta,\alpha}.$$

The condition numbers measures the sensitivity of the inverse of a matrix to perturbations in the data.

1.1.6 Floating Point Numbers

A floating point number system $F \subset \mathbb{R}$ is a subset of the real numbers of the form

$$y = \pm m \times \beta^{e-t},$$

for mantissa m, base β, precision t and exponent range $e_{\min} \leq e \leq e_{max}$. This representation is employed by all modern computer systems.

For $fl(x)$ denoting an element of F that is nearest to x, we have the following theorem [29].

Theorem 1.1.1 *If $x \in \mathbb{R}$ lies in the range of F then*

$$fl(x) = x(1+\sigma), \quad |\sigma| < u = \frac{1}{2}\beta^{1-t}. \quad \square$$

Thus the real number x can be approximated by an element in F with a relative error of u, which we call the *unit roundoff*.

An operation involving floating point numbers, such as addition, subtraction, multiplication and division we call a *floating point operation* or *flop*.

Throughout this thesis we quote only the highest order for flop counts. That is, if an algorithm requires $3n^3 + 2n^2$ flops to compute, we say only that this requires $3n^3$ flops. This as because for large n, usually the case we are interested in, the $2n^2$ term is not significant compared to the higher order term.

We say that $fl(x)$ *overflows* if

$$|fl(x)| > \max\{|y| : y \in F\}$$

and *underflows* if

$$0 < |fl(x)| < \min\{|y| : 0 \neq y \in F\}.$$

If the number system consists of all real numbers we have the concept of *exact arithmetic*, since every real number can be represented, and the unit roundoff is zero.

A phenomenon that can arise in floating point arithmetic is *cancellation error*. It is the extreme loss of significant digits when two floating point numbers, that are nearly equal, are subtracted. See [29, Sec. 1.7] for an example. In practice computations can often be performed in an order that reduces or avoids cancellation error.

1.2 Eigenvalue Problems

1.2.1 The Standard Eigenvalue Problem

If $A \in \mathbb{C}^{n\times n}$, $x \neq 0 \in \mathbb{C}^n$ and $\lambda \in \mathbb{C}$, then we call the problem

$$Ax = \lambda x \tag{1.2.1}$$

the standard eigenvalue problem. The scalar λ is an *eigenvalue* and x is a corresponding *eigenvector*.

If we write (1.2.1) as

$$(A - \lambda I)x = 0,$$

then we see that x is in the null space of the matrix $A - \lambda I$, which therefore has to be nonempty. Equivalently

$$\det(A - \lambda I) = 0,$$

and this degree n polynomial is called the *characteristic polynomial*. We can write this in factored form

$$\det(A - \lambda I) = (\lambda - \lambda_1)(\lambda - \lambda_2)\dots(\lambda - \lambda_n),$$

where the λ_i are the eigenvalues of (1.2.1). Some of the λ_i may be repeated, so we write

$$\det(A - \lambda I) = (\lambda - \widetilde{\lambda}_1)^{m_1}(\lambda - \widetilde{\lambda}_2)^{m_2}\dots(\lambda - \widetilde{\lambda}_k)^{m_k},$$

where the $\widetilde{\lambda}_i$ are distinct and m_i is the *algebraic multiplicity* of $\widetilde{\lambda}_i$, with $\sum_i m_i = n$. The eigenvalue $\widetilde{\lambda}_i$ also has a *geometric multiplicity*, which is the maximum number of linearly independent eigenvectors, or, equivalently, the dimension of the null space of $A - \widetilde{\lambda}_i I$.

An eigenvalue with algebraic multiplicity 1 is a *simple* eigenvalue. If λ_i has algebraic multiplicity greater than its geometric multiplicity then the matrix and the eigenvalue we describe as *defective*. Defective matrices have less than n linearly independent eigenvectors.

1.2.2 The Hermitian Eigenvalue Problem

If $A \in \mathbb{C}^{n\times n}$ is Hermitian (or real and symmetric) we call the eigenvalue problem the Hermitian (symmetric) eigenvalue problem.

Theorem 1.2.1 (Schur Decomposition) *If $A \in \mathbb{C}^{n\times n}$ then there exists a unitary matrix $Q \in \mathbb{C}^{n\times n}$ such that*

$$QAQ^* = U,$$

where U is upper triangular. □

If we apply Theorem 1.2.1 to Hermitian A and take complex conjugates of U then

$$U^* = (QAQ^*)^* = (Q^*)^* A^* Q^* = QAQ^* = U.$$

So U is both upper triangular and Hermitian and is therefore diagonal.

If we write $AQ = QU$, then from (1.2.1), the columns of Q are linearly independent eigenvectors and the U_{ii} are eigenvalues. Since $U^* = U$ then the eigenvalues of the Hermitian (or real and symmetric) eigenvalue problem are real. We then have the following theorem.

Theorem 1.2.2 (Spectral Decomposition) *If $A \in \mathbb{C}^{n\times n}$ is Hermitian then there exists a unitary matrix Q and a diagonal matrix $\Lambda \in \mathbb{R}^{n\times n}$ such that*

$$QAQ^* = \Lambda. \quad \square$$

1.2.3 Deflation

The determinant of a block upper triangular matrix is equal to the product of the determinants of the individual diagonal blocks. Then if we seek the eigenvalues of a block upper triangular matrix U, we have

$$\det(U - \lambda I) = \prod_i \det(U_{ii} - \lambda I).$$

Thus the problem can split into smaller subproblems which can be solved independently. We call this process *deflation*.

1.2.4 Similarity

If $A \in \mathbb{C}^{n\times n}$ and $Y \in \mathbb{C}^{n\times n}$ is nonsingular, then $B = YAY^{-1}$ is said to be *similar* to A.

Theorem 1.2.3 *If $A \in \mathbb{C}^{n\times n}$ and $B \in \mathbb{C}^{n\times n}$ are similar, then they have the same eigenvalues.*

Proof. If λ is an eigenvalue of A with corresponding eigenvector x, then for nonsingular $Y \in \mathbb{C}^{n\times n}$

$$B(Yx) = YAY^{-1}Yx = YAx = \lambda(Yx),$$

so λ is an eigenvalue of B with corresponding eigenvector $Yx \neq 0$. □

1.2.5 Definiteness

A Hermitian matrix is said to be *positive definite* if

$$x^*Ax > 0, \quad \text{for any } x \neq 0,$$

and *positive semidefinite* if the inequality is not strict.

Theorem 1.2.4 *If $A \in \mathbb{C}^{n\times n}$ is Hermitian and positive definite all its eigenvalues are real and positive.*

Proof. Since A is Hermitian it has real eigenvalues and if x is an eigenvector of A then

$$0 < x^*Ax = x^*\lambda x = \lambda(x^*x) \Rightarrow \lambda > 0. \qquad \square$$

Similarly, $\lambda \geq 0$ if A is *positive semidefinite*.

We define *negative definiteness* and *negative semidefiniteness* in a analogous way. If a matrix is neither positive semidefinite nor negative semidefinite we describe it as *indefinite*.

1.2.6 The Generalized Eigenvalue Problem

The generalized eigenvalue problem is of the form

$$Ax = \lambda Bx, \tag{1.2.2}$$

where $A, B \in \mathbb{C}^{n\times n}$, $\lambda \in \mathbb{C}$ is an eigenvalue and $x \neq 0 \in \mathbb{C}^n$ a corresponding eigenvector. Equivalently, λ is an eigenvalue if $\det(A - \lambda B) = 0$.

If B is nonsingular we can write $B^{-1}Ax = \lambda x$ or $AB^{-1}y = \lambda y,\ y = Bx$ which are standard eigenvalue problems. We note that $AB^{-1} = B(B^{-1}A)B^{-1}$ is similar to $B^{-1}A$ so the problems share the same eigenvalues.

We call the matrix $A - \lambda B$ a *matrix pencil*, and an eigenvector x is a null vector of the pencil. This matrix pencil is *nonregular* if $\det(A - \lambda B) = 0$ for all λ, else it is *regular*.

We can write (1.2.2) as

$$\beta Ax = \alpha Bx, \quad \lambda = \alpha/\beta. \tag{1.2.3}$$

This is a convenient notation that allows us to describe all eigenvalues of (1.2.2). If $\beta = 0$ we describe the eigenvalue as *infinite*. If B is singular then the null vectors of B will be eigenvectors corresponding to infinite eigenvalues. If A and B share a common null space, $Ax = Bx = 0, x \neq 0$, then $(\beta A - \alpha B)x = 0$ for all α and β and the pencil is nonregular. A non-empty intersection of the null spaces of A and B is a sufficient but not necessary condition for the pencil to be nonregular: for example A and B in the pencil

$$A - \lambda B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \lambda \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

have no null vectors in common but the pencil is nonregular.

If $\beta \neq 0$ then the eigenvalue is *finite*, with $\alpha = 0$ corresponding to a zero eigenvalue. We note that if A is singular then the null vectors of A are

eigenvectors corresponding to zero eigenvalues, provided they are not also in the null space of B, as then α and β could take any value.

As with the standard eigenvalue problem, for a single eigenvalue we describe the maximum number of corresponding linearly independent eigenvectors as its *geometric multiplicity*. And the number of times an eigenvalue is repeated we call its *algebraic multiplicity*. An eigenvalue whose algebraic multiplicity exceeds its geometric multiplicity is a *defective eigenvalue* and a pencil containing one or more defective eigenvalues is a *defective matrix pencil*.

We can determine the α and β in (1.2.3) by the following theorem.

Theorem 1.2.5 (Generalized Schur Decomposition) *If $A, B \in \mathbb{C}^{n\times n}$ then there exist unitary matrices Q and Z such that*

$$Q^*AZ = \widetilde{A}, \qquad Q^*BZ = \widetilde{B}, \tag{1.2.4}$$

where $\widetilde{A}$ and $\widetilde{B}$ are upper triangular. □

Applying Theorem 1.2.5 to our pencil we have

$$\det(A - \lambda B) = \det(QZ^*) \prod_{i=1}^{n} (a_{ii} - \lambda b_{ii}), \tag{1.2.5}$$

where a_{ii} and b_{ii} are the diagonal entries of $\widetilde{A}$ and $\widetilde{B}$ respectively. If $a_{ii} = b_{ii} = 0$ for at least one i, then the pencil is nonregular. Conversely, the pencil is only nonregular if there is such an a_{ii} and b_{ii}.

If the pencil is regular then (1.2.5) shows that there are n eigenvalues that are either finite or infinite.

1.2.7 The Symmetric Semidefinite Generalized Eigenvalue Problem

If for

$$\beta Ax = \alpha Bx, \quad \lambda = \alpha/\beta, \tag{1.2.6}$$

$A, B \in \mathbb{R}^{n\times n}$ are symmetric and one is semidefinite we call the problem the symmetric semidefinite generalized eigenvalue problem.

1.2.8 Kronecker Canonical Forms for Matrix Pencils

The Kronecker Canonical Form (KCF) is given by the following theorem.

Theorem 1.2.6 (The Kronecker Canonical Form) *If $A, B \in \mathbb{C}^{m\times n}$ then there exist matrices $P \in \mathbb{C}^{m\times m}$ and $Q \in \mathbb{C}^{n\times n}$ such that*

$$P^{-1}(A-\lambda B)Q = \begin{bmatrix} A_1-\lambda B_1 & & & \\ & A_2-\lambda B_2 & & \\ & & \ddots & \\ & & & A_r-\lambda B_r \end{bmatrix} \tag{1.2.7}$$

is block diagonal, where $r \le m$ and the blocks $A_i-\lambda B_i$ take one of the following four forms

$$J_j(\gamma) = \begin{bmatrix} \gamma-\lambda & 1 & & \\ & \ddots & & \\ & & & 1 \\ & & & \gamma-\lambda \end{bmatrix} \in \mathbb{C}^{j\times j}, \quad j = 1, 2\ldots$$

$$N_j = \begin{bmatrix} 1 & -\lambda & & \\ & \ddots & & \\ & & & -\lambda \\ & & & 1 \end{bmatrix} \in \mathbb{C}^{j\times j}, \quad j = 1, 2\ldots$$

$$L_j = \begin{bmatrix} -\lambda & 1 & & \\ & \ddots & & \\ & & -\lambda & 1 \end{bmatrix} \in \mathbb{C}^{j\times(j+1)}, \quad j = 0, 1, 2\ldots$$

$$L_j^T = \begin{bmatrix} -\lambda & & \\ 1 & & \\ & \ddots & \\ & & -\lambda \\ & & 1 \end{bmatrix} \in \mathbb{C}^{(j+1)\times j}, \quad j = 0, 1, 2 \ldots . \quad \square$$

The $J_j(\gamma)$ blocks give the finite eigenvalues γ. The number of times a $J_j(\gamma)$ block appears gives the geometric multiplicity of the eigenvalue. The sum of the orders of these blocks gives the algebraic multiplicity for a particular γ.

The N_j blocks give the infinite eigenvalues, with the number of N_j blocks giving the geometric multiplicity for infinite eigenvalues.

The L_j blocks are called *right singular blocks*, and the L_j^T blocks are called *left singular blocks*. Together they define the *nonregular structure* of the matrix pencil. Note an L_0 block has zero rows and an L_0^T zero columns. The interpretation of this is for $X \in \mathbb{C}^{n\times n}$ and $Y \in \mathbb{C}^{(n-1)\times(n-1)}$

$$X = \begin{bmatrix} L_0 & & \\ & Y & \\ & & L_0^T \end{bmatrix} = \begin{bmatrix} & Y \\ 0 & \end{bmatrix}.$$

Cao [11] shows that the KCF of a pencil when A and B are symmetric and B is positive semidefinite consists only of blocks of the following forms:

$$\begin{aligned} &J_1(\gamma) = \gamma - \lambda, && N_1 = 1 - 0 \\ &N_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \lambda \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, && L_0 \text{ and } L_0^T. \end{aligned}$$

1.3 Matrix Transformations

1.3.1 Congruence Transformations

The transformation $Y \leftarrow Q^*XQ$, with Q nonsingular, we call a *congruence transformation*. These transformations have two important properties when

applied to a matrix pencil. Firstly, symmetry is preserved, taking the conjugate transpose of the transformed pencil we have

$$(Q^*(A-\lambda B)Q)^* = Q^*(A-\lambda B)^*(Q^*)^* = Q^*(A-\lambda B)Q.$$

Also, eigenvalues are preserved, the pencils are equivalent.

Theorem 1.3.1 *Let $A, B \in \mathbb{C}^{n\times n}$ be Hermitian and Q be nonsingular. Then if λ is an eigenvalue of the matrix pencil $A-\lambda B$ then it is also an eigenvalue of the transformed pencil $Q^*(A-\lambda B)Q$. Also, if x is an eigenvector corresponding to λ in $A-\lambda B$ then $y = Q^{-1}x$ is the corresponding eigenvector in the transformed pencil.*

Proof. If λ and x are an eigenvalue and corresponding eigenvector of the pencil $A-\lambda B$ then

$$Q^*AQ(Q^{-1}x) = Q^*Ax = \lambda Q^*Bx = \lambda Q^*BQ(Q^{-1}x)$$

and λ and $Q^{-1}x$ are an eigenvalue and corresponding eigenvector of the transformed pencil. □

1.3.2 Unitary Transformations

If $Q \in \mathbb{C}^{n\times n}$ is unitary (or real and orthogonal) we call the transformation $Y \leftarrow Q^*XQ$ a *unitary (or orthogonal) similarity transformation.* This transformation clearly has the same properties as a congruence transformation.

The fact that the Frobenius and 2-norm are unitarily invariant has advantages for unitary transformations. Consider a matrix X which has errors in the form of a perturbation matrix, E. If we apply the orthogonal transformation

$$Y + F \leftarrow Q^T(X+E)Q,$$

then

$$\|F\|_2 = \|Q^T E Q\|_2 = \|E\|_2 \quad \text{and} \quad \|F\|_F = \|Q^T E Q\|_F = \|E\|_F.$$

The errors in the transformed matrix have the same norm as the error in the original matrix.

1.4 The QR Factorization

For $A \in \mathbb{R}^{m\times n}$ the QR factorization is given by

$$A = QR, \tag{1.4.1}$$

where $Q \in \mathbb{R}^{m\times m}$ is orthogonal and $R \in \mathbb{R}^{m\times n}$ is upper trapezoidal.

1.4.1 Computing the QR Factorization

We compute the QR factorization of $A \in \mathbb{R}^{m\times n}$, $m \geq n$ and of full rank, by applying an orthogonal transformation matrix Q^T so that

$$Q^T A = R,$$

and Q is the product of orthogonal matrices chosen to transform A to be the upper triangular matrix R.

One method uses *Givens matrices* to introduce zeros below the diagonal one element at a time. A Givens matrix, $G(i,j) \in \mathbb{R}^{m\times m}$, is of the form

$$G(i,j) = \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} i & & j & \end{array} \\ \begin{bmatrix} I & & & \\ & c & s & \\ & & I & \\ & -s & c & \\ & & & I \end{bmatrix} \end{array}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$ for some θ, and is therefore orthogonal.

For $x \in \mathbb{R}^n$, if we set

$$c = \frac{x_i}{\sqrt{x_i^2 + x_j^2}}, \quad s = \frac{-x_j}{\sqrt{x_i^2 + x_j^2}},$$

then for $G(i,j)^T x = y$,

$$y_k = \begin{cases} cx_i - sx_j & k = i, \\ 0 & k = j, \\ x_k & k \neq i, j, \end{cases}$$

so only the ith and jth elements are affected. We can compute c and s by the following algorithm. All the algorithms in this thesis are presented in a MATLAB-like pseudo code.

Algorithm 1.4.1 *This function returns scalars c and s such that* $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} d \\ 0 \end{bmatrix}$, *where a, b, and d are scalars, and $s^2 + c^2 = 1$.*

```
function [c, s] = givens(a, b)
if b = 0
    c = 1
    s = 0
else
    if abs(b) ≥ abs(a)
        t = −a/b
        s = 1/√(1 + t²)
        c = st
    else
        t = −b/a
        c = 1/√(1 + t²)
        s = ct
    end
end
```

Here the computation of c and s has been rearranged to avoid possible overflow.

Now to transform A to an upper trapezoidal matrix we require a Givens matrix for each subdiagonal element of A, and apply each one in a suitable order such as

$$G(n, n+1)^T \dots G(m-1, m)^T G(1, 2)^T \dots G(m-1, m)^T A = Q^T A = R,$$

which is a QR factorization. For example for $m = 6$ and $n = 4$, with a $+$ denoting a nonzero entry:

$$A = \begin{bmatrix} + & + & + & + \\ + & + & + & + \\ + & + & + & + \\ + & + & + & + \\ + & + & + & + \\ + & + & + & + \end{bmatrix} \overset{G(5,6)^T}{\rightarrow} \begin{bmatrix} + & + & + & + \\ + & + & + & + \\ + & + & + & + \\ + & + & + & + \\ + & + & + & + \\ 0 & + & + & + \end{bmatrix} \overset{G(4,5)^T}{\rightarrow} \dots$$

$$\overset{G(1,2)^T}{\rightarrow} \begin{bmatrix} + & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \end{bmatrix} \overset{G(5,6)^T}{\rightarrow} \begin{bmatrix} + & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \\ 0 & 0 & + & + \end{bmatrix} \overset{G(4,5)^T}{\rightarrow} \dots$$

$$\overset{G(4,5)^T}{\rightarrow} \begin{bmatrix} + & + & + & + \\ 0 & + & + & + \\ 0 & 0 & + & + \\ 0 & 0 & 0 & + \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

This strategy is adopted in the following algorithm.

Algorithm 1.4.2 *This algorithm computes the R factor of a Givens QR factorization of a matrix $A \in \mathbb{R}^{m \times n}$, $m \geq n$, overwriting the upper triangular part of A with the nonzero elements of R.*

for $j = 1\colon n$
  for $i = m\colon -1\colon j+1$
    $[c, s] = \mathbf{givens}(A(i-1, j), A(i, j))$
    $A(i-1, j) = cA(i-1, j) - sA(i, j)$
    $A(i-1\colon i, j+1\colon n) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T A(i-1\colon i, j+1\colon n)$
  end
end

Note we do not compute quantities we know are zero. Also, the matrix Q is not formed but applied implicitly. We can store c and s and if Q is required to multiply another matrix, say, we can do this in a similar way to Algorithm 1.4.2. It is possible to encode c and s in a single scalar (see [24, Sec. 5.1.11]), which could then be stored in the eliminated a_{ij}.

The primary use of Givens matrices is to eliminate particular elements in a matrix. A more efficient approach for a QR factorization is to use *Householder matrices* which introduce zeros in all the subdiagonal elements of a column simultaneously.

Householder matrices, $H \in \mathbb{R}^{n\times n}$, are of the form

$$H = I - \tau v v^T, \quad \tau = \frac{2}{v^T v},$$

where the *Householder vector*, $v \in \mathbb{R}^n$, is nonzero. It is easy to see that H is symmetric and orthogonal. If y and z are distinct vectors such that $\|y\|_2 = \|z\|_2$ then there exists an H such that

$$Hy = z.$$

We can determine a Householder vector such that

$$H \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix},$$

where $x \in \mathbb{R}^{n-1}$, and α and β are scalars. By setting

$$v = \begin{bmatrix} \alpha \\ x \end{bmatrix} \pm \left\| \begin{bmatrix} \alpha \\ x \end{bmatrix} \right\|_2 e_1. \tag{1.4.2}$$

We then have

$$H \begin{bmatrix} \alpha \\ x \end{bmatrix} = \mp \left\| \begin{bmatrix} \alpha \\ x \end{bmatrix} \right\|_2 e_1,$$

that is

$$\beta = \mp \left\| \begin{bmatrix} \alpha \\ x \end{bmatrix} \right\|_2 .$$

In we choose the sign in (1.4.2) to be negative then β is positive. However, if $[\, \alpha \quad x^T \,]$ is close to a positive multiple of e_1, then this can give large cancellation error. So we use the formula [40]

$$v_1 = \alpha - \| \, [\, \alpha \quad x^T \,] \, \|_2 = \frac{\alpha^2 - \| \, [\, \alpha \quad x^T \,] \, \|_2^2}{\alpha + \| \, [\, \alpha \quad x^T \,] \, \|_2} = \frac{-\|x\|_2^2}{\alpha + \| \, [\, \alpha \quad x^T \,] \, \|_2}$$

to avoid this in the case when $\alpha > 0$. This is adopted in the following algorithm.

Algorithm 1.4.3 *For $\alpha \in \mathbb{R}$ and $x \in \mathbb{R}^{n-1}$ this function returns a vector $v \in \mathbb{R}^{n-1}$ and a scalar τ such that $\tilde{v} = \begin{bmatrix} 1 \\ v \end{bmatrix}$ is a Householder vector, scaled so $\tilde{v}(1) = 1$ and $H = \left[I - \tau \begin{bmatrix} 1 \\ v \end{bmatrix} [\, 1 \quad v^T \,] \right]$ is orthogonal, with $H \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}$, where $\beta \in \mathbb{R}$.*

function $[v, \tau] =$ **householder**(α, x)
$s = \|x\|_2^2$
$v = x$
if $s = 0$
    $\tau = 0$
else
    $t = \sqrt{\alpha^2 + s}$
    % choose sign of v
    if $\alpha \leq 0$
        $v\_one = \alpha - t$
    else

```
        v_one = −s/(α + t)
    end
    τ = 2v_one²/(s + v_one²)
    v = v/v_one
end
```

Here we have normalized v so $v_1 = 1$ and the *essential* part of the Householder vector, $v(2\colon n)$, can be stored in x.

Thus if we apply n Householder matrices, H_j, to introduce zeros in the subdiagonal columns one by one, we have the QR factorization

$$H_n \ldots H_1 A = Q^T A = R,$$

and the H_j are such that their vectors v_j are of the form

$$\begin{aligned} v_j(1\colon j-1) &= 0, \\ v_j(j) &= 1, \\ v_j(j+1\colon m)\,\colon & \quad \text{as } v \text{ in Algorithm 1.4.3.} \end{aligned}$$

This leads to the following Householder QR factorization algorithm.

Algorithm 1.4.4 *This algorithm computes a Householder QR factorization of a matrix $A \in \mathbb{R}^{m\times n}$, $m \geq n$, overwriting the upper triangular part of A with the nonzero elements of R. Q is not formed explicitly. The essential part of the Householder vectors are stored in the subdiagonal of A.*

```
for j = 1: n
[v, τ(j)] = householder(A(j, j), A(j + 1: m, j))
    A(j, j) = A(j, j) − τ(j)A(j, j) + v^T A(j + 1: m, j))
    if j < n
        A(j: m, j + 1: n) = A(j: m, j + 1: n)−
                τ(j) [ 1   v^T ]^T ([ 1   v^T ] A(j: m, j + 1: n))
    end
    A(j + 1: m, j) = v
end
```

The algorithm requires $2n^2(m-n/3)$ flops. The essential part of the Householder vectors are stored in the subdiagonal, and we refer to Q being in *factored form.* If Q is required to be applied to another matrix, it can be applied as in Algorithm 1.4.4.

If Q is to be formed explicitly we can do so with the *backward accumulation* method by computing

$$(H_1 \ldots (H_{n-2}(H_{n-1}H_n))).$$

This exploits the fact that the leading $(j-1)$-by-$(j-1)$ part of H_j is the identity.

Algorithm 1.4.5 *This algorithm forms the orthogonal matrix $Q \in \mathbb{R}^{m\times m}$ of the QR factorization as Algorithm 1.4.4, with the essential part of the Householder vectors stored in the subdiagonal of A and the vector τ from Algorithm 1.4.4.*

$Q = I$
for $j = n\colon -1\colon 1$
  $v = A(j+1\colon m, j)$
  $Q(j\colon m, j\colon m) = Q(j\colon m, j\colon m) -$
    $\tau(j)\,[\,1 \quad v^T\,]^T\,([\,1 \quad v^T\,]\,Q(j\colon m, j\colon m))$
end

1.4.2 The Blocked QR Factorization

A *blocked algorithm* acts on a blocked matrix. We discuss the benefits of this in Section 1.6.1. We derive a blocked Householder QR factorization as follows [45].

Theorem 1.4.1 *We can write the product of p Householder matrices, $H_i = I - \tau_i v_i v_i^T$, as*

$$H_1 H_2 \ldots H_p = I - VTV^T,$$

where

$$V = [\,v_1 \quad v_2 \quad \ldots \quad v_p\,] = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix},$$

and $V_1 \in \mathbb{R}^{p\times p}$ is lower triangular and T is upper triangular.

Proof. The proof is inductive. For $p = 1$ we have $T = \tau_1$ which is an upper triangular matrix. Now assuming it is true for $p-1$, we have

$$H_1 H_2 \dots H_{p-1} = I - VTV^T,$$

so

$$\begin{aligned} H_1 H_2 \dots H_{p-1} H_p &= (I - VTV^T)(I - \tau_p v_p v_p^T) \\ &= I - \tau_p v_p v_p^T - VTV^T + \tau_p VTV^T v_p v_p^T \\ &= I - \begin{bmatrix} V & v_p \end{bmatrix} \begin{bmatrix} TV^T - \tau_p TV^T v_p v_p^T \\ \tau_p v_p^T \end{bmatrix} \\ &= I - \begin{bmatrix} V & v_p \end{bmatrix} \begin{bmatrix} T & -\tau_p TV^T v_p \\ 0 & \tau_p \end{bmatrix} \begin{bmatrix} V^T \\ v_p^T \end{bmatrix}. \quad \square \end{aligned}$$

With this representation of the Householder vectors we can derive a blocked algorithm. At the kth step we apply Algorithm 1.4.4 to the first p columns of A only, for some block size p. When the current block has been factorized we apply $I - VT^TV^T$, the product of the p Householder matrices, to update the trailing matrix. This leads to the following algorithm.

Algorithm 1.4.6 *This algorithm computes $Q^TA = R \in \mathbb{R}^{m\times n}$, $m \geq n$, using a blocked method with block size n_b, overwriting the upper triangular part of A with the nonzero elements of R.*

```
for k = 1:n_b:n
    % Check for the last column block
    jb = min(n_b, n - k + 1)
    for j = k:k + jb - 1
        [v, τ(j)] = householder(A(j, j), A(j + 1:m, j))
        % Update trailing part of current block only
        A(j, j) = A(j, j) − τ(j)A(j, j) + v^T A(j + 1:m, j))
        if j < k + jb − 1
            A(j:m, j + 1:k + jb − 1) = A(j:m, j + 1:k + jb − 1)−
```

$$\tau(j)\,[\,1 \quad v^T\,]\,([\,1 \quad v^T\,]\,A(j\colon m, j+1\colon k+jb-1))$$

end

$V(j,j) = 1$

$V(j+1\colon m, j) = v$

end

% If we are not in last block column

% build T and update trailing matrix

if $k + jb \leq n$

for $j = k\colon k+jb-1$

% Build T

if $j = k$

$T(1,1) = \tau(j)$

else

$T(1\colon j-k, j-k+1) = -\tau(j)T(1\colon j-k, 1\colon j-k)$

$*V(j\colon m, k\colon j-1)^T V(j\colon m, j)$

$T(j-k+1, j-k+1) = \tau(j)$

end

end

% Update trailing matrix

$A(k\colon m, j+1\colon n) = A(k\colon m, j+1\colon n)$

$-V(k\colon m, k\colon k+jb-1)T(1\colon jb, 1\colon jb)^T(V(k\colon m, k\colon k+jb-1)^T$

$*A(k\colon m, j+1\colon n))$

end

end

1.5 Rank Revealing Decompositions

A rank-revealing decomposition (RRD) for a symmetric matrix $A \in \mathbb{R}^{n\times n}$ takes the form

$$A = XDX^T,$$

where $X \in \mathbb{R}^{n\times n}$ is well conditioned and $D \in \mathbb{R}^{n\times n}$ is diagonal. For a general matrix $C \in \mathbb{R}^{m\times n}$ an RRD takes the form

$$C = XGZ,$$

where $X \in \mathbb{R}^{m\times m}$ and $Z \in \mathbb{R}^{n\times n}$ are well conditioned and $G \in \mathbb{R}^{m\times n}$ is trapezoidal or diagonal. The diagonal elements of D and G are arranged in decreasing order of absolute value.

We let η denote a tolerance used to define negligibility of matrix elements. In floating point arithmetic η will usually be some multiple of the unit roundoff, u. That is, for

$$D = \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix}, \quad \text{and} \quad G = \begin{bmatrix} G_{11} & G_{12} \\ 0 & G_{22} \end{bmatrix},$$

then if

$$\|D_2\| \le \eta \quad \text{and} \quad \|G_{22}\| \le \eta,$$

for some norm, we set them to zero. If the order of D_2 and G_{22} is as large as possible, the dimension of D_1 or G_{11} is the rank of the matrix. In exact arithmetic we could set $\eta = 0$.

1.5.1 The Pivoted QR Factorization

A *pivoted* factorization involves the swapping of rows and/or columns to order elements in the factors. This helps to decide rank. The pivoting is represented in the factorization by a permutation matrix.

We can factor a rank deficient matrix, $A \in \mathbb{R}^{m\times n}$, by using a pivoted QR factorization

$$A = QRP^T, \tag{1.5.1}$$

where $Q \in \mathbb{R}^{m\times m}$ is orthogonal, $P \in \mathbb{R}^{n\times n}$ is a permutation matrix and $R \in \mathbb{R}^{m\times n}$ is of the form

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix},$$

where $R_{11} \in \mathbb{R}^{r\times r}$ is upper triangular and r is the rank of A. R satisfies

$$r_{kk}^2 \ge \sum_{i=k}^{\min(j,r)} r_{ij}^2, \quad j = k+1\colon n, \quad k = 1\colon r,$$

and

$$|r_{11}| \geq |r_{22}| \geq \cdots \geq |r_{rr}| > 0.$$

The column pivoting strategy is, at the kth stage, to ensure

$$\|a_k^{(k)}(k\colon m)\|_2 = \max_{j \geq k} \|a_j^{(k)}(k\colon m)\|_2,$$

that is we permute the column with the largest norm to the leading position from the trailing matrix $A^{(k-1)}(A(k\colon m, k\colon n)$.

Note that since $\|Qx\|_2^2 = \|x\|_2^2$ for any orthogonal matrix Q, then for $Qx = y \in \mathbb{R}^n$

$$\|y(2\colon n)\|_2^2 = \|x\|_2^2 - y_1^2$$

and so the new column norms used for pivoting need not be completely recalculated.

The algorithm requires $4(mnr + m^2 r) - 2r^2(3m/2 + n) + 5/3r^3$ flops. For references for rank revealing QR algorithms see [14] and [29].

1.5.2 The Complete Orthogonal Decomposition

After the pivoted QR factorization of $A \in \mathbb{R}^{m \times n}$ of rank r we have

$$Q^T A P = R = \begin{array}{c} \\ r \\ m-r \end{array} \begin{array}{c} \begin{array}{cc} r & n-r \end{array} \\ \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \end{array},$$

where R_{11} is upper triangular. We can further reduce R by eliminating R_{12} with Householder matrices.

We find r Householder matrices such that

$$H_r \ldots H_1 \begin{bmatrix} R_{11}^T \\ R_{12}^T \end{bmatrix} = \begin{bmatrix} \overline{R}_{11}^T \\ 0 \end{bmatrix},$$

where $\overline{R}_{11}$ is upper triangular. Then we have

$$\begin{aligned} A &= Q(RH_1^T \dots H_r^T)H_r \dots H_1 P^T \\ &= Q \begin{bmatrix} \overline{R}_{11} & 0 \\ 0 & 0 \end{bmatrix} V, \\ &= QUV, \end{aligned} \tag{1.5.2}$$

where $V = H_r \dots H_1 P^T$, which is the *complete orthogonal decomposition.*

The algorithm requires $4(mnr + m^2 r) - r^2(3m + 7n) - 5/3r^3 + 4/3n^3 - 4n^2 r$ flops.

1.5.3 The Spectral Decomposition

For symmetric $A \in \mathbb{R}^{n \times n}$ we can perform the spectral decomposition (Theorem 1.2.2)

$$A = Q\Lambda Q^T, \tag{1.5.3}$$

where Q is orthogonal and contains the eigenvectors of A and Λ is diagonal and contains the eigenvalues of A, which we will order according to

$$|\lambda_{11}| \geq |\lambda_{22}| \geq \dots \geq |\lambda_{rr}| > |\lambda_{r+1,r+1}| = \dots = 0,$$

where r is the rank of A.

There are many ways algorithmically that we can compute a spectral decomposition. The most popular method for symmetric matrices, employed by LAPACK [1] and MATLAB [36], is to first reduce the matrix A to tridiagonal form by a symmetric Householder QR factorization so that

$$Q_1^T A Q_1 = T,$$

where T is a tridiagonal (and symmetric) matrix. If any off diagonal elements are zero the problem can be deflated into smaller subproblems, if not it is said

to be *unreduced*. The unreduced matrix is then diagonalized by a *shifted* QR iteration to give

$$Q_2^T T Q_2 = \Lambda,$$

where $Q_2 Q_1 = Q$ in (1.5.3) .

The shift is a process whereby convergence is accelerated by computing the QR factorization of $A - \mu I$ instead of A, where μ is an approximate eigenvalue, giving the iteration

$$\begin{aligned} T^{(k)} - \mu I &= Q_2 R \\ T^{(k+1)} &= R Q_2 + \mu I, \end{aligned}$$

$T^{(k+1)}$ is similar to $T^{(k)}$ since $RQ_2 + \mu I = Q_2^T (Q_2 R + \mu I) Q_2 = Q_2^T T^{(k)} Q_2$, and the $T^{(k+1)}$'s converge to Λ. See [24, Alg. 8.3.3] for details and discussion on the choice of μ.

The algorithm tends to order the eigenvalues in Λ algebraically. Ordering the eigenvalues by absolute size is achieved by a trivial permutation of Λ and correspondingly Q. A decision on rank can then be made by deciding which values of λ_i can be taken as zero.

The algorithm requires an average of $9n^3$ flops to compute the eigenvalues and Q, but depends on the amount of iterations required.

The divide and conquer algorithm [19] gives an alternative way of solving the tridiagonal problem. It repeatedly divides the tridiagonal matrix into two halves, solves the eigenproblems of each half, and joins the solutions together.

The eigenvalues and eigenvectors can be computed with $4n^3$ flops. This 'best' figure we use in this thesis for comparing flop counts with other algorithms.

1.5.4 The Singular Value Decomposition

Theorem 1.5.1 (Singular Value Decomposition (SVD)) *For $A \in \mathbb{R}^{n\times n}$ there exist orthogonal matrices*

$$\begin{aligned} U &= [u_1, \ldots, u_m] \in \mathbb{R}^{m\times m} \quad \textit{and} \\ V &= [v_1, \ldots, v_n] \in \mathbb{R}^{n\times n} \end{aligned}$$

such that

$$U^T A V = \mathrm{diag}(\sigma_1, \ldots \sigma_p) \in \mathbb{R}^{m\times n}, \tag{1.5.4}$$

where $p = \min(m, n)$ and

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0. \quad \square$$

If $A \in \mathbb{C}^{n\times n}$ then the same relation applies with unitary U and V.

The σ_i are the *singular values* of A and the vectors u_i and v_i are the ith *left singular vectors* and the ith *right singular vectors* respectively.

In exact arithmetic if

$$\sigma_1 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_p = 0,$$

then the rank of A is r.

The Golub-Reinsch algorithm for the SVD requires $4m^2n + 8mn^2 + 9n^3$ flops, we use this figure for comparing flop counts with other algorithms. For further details see [24].

1.5.5 The LDL^T Factorization

The LDL^T factorization of a symmetric matrix $A \in \mathbb{R}^{n\times n}$ takes the form

$$PAP^T = LDL^T, \tag{1.5.5}$$

where $P \in \mathbb{R}^{n\times n}$ is a permutation matrix, $L \in \mathbb{R}^{n\times n}$ is unit lower triangular, and $D \in \mathbb{R}^{n\times n}$ is block diagonal, with blocks having dimension 1 or 2.

The factorization is computed as follows. Suppose we can find a permutation Π such that

$$\Pi A \Pi^T = \begin{array}{c} \\ s \\ n-s \end{array} \begin{array}{c} \begin{array}{cc} s & n-s \end{array} \\ \begin{bmatrix} E & C^T \\ C & B \end{bmatrix} \end{array},$$

where E is of order 1 or 2 and nonsingular. We can then factorize

$$\Pi A \Pi^T = \begin{bmatrix} I_s & 0 \\ CE^{-1} & I_{n-s} \end{bmatrix} \begin{bmatrix} E & 0 \\ 0 & B - CE^{-1}C^T \end{bmatrix} \begin{bmatrix} I_s & E^{-1}C^T \\ 0 & I_{n-s} \end{bmatrix}.$$

Repeating the process recursively on $B - CE^{-1}C^T$ gives the factorization. The algorithm requires $n/3$ flops plus the permutation overhead.

The permutations Π, and whether $s = 1$ or 2 can be determined by several different methods. Two such methods are:

- *Partial Pivoting* due to Bunch and Kauffman [8], where at each stage only two columns need to be searched and only $O(n^2)$ comparisons are required.

- *Rook Pivoting* due to Ashcroft, Grimes and Lewis [2], which is similar to partial pivoting but has an iterative stage.

See [29] for further details and an error analysis.

We can reduce the block diagonal matrix D to diagonal form by applying a spectral decomposition to each 2-by-2 block in D, if

$$D = \operatorname{diag}(D_{11}, D_{22}, \ldots, D_{pp})$$

for p blocks and D_{ii} is either a single entry or a 2-by-2 block, then we factor the 2-by-2 blocks

$$D_{ii} = Q_{ii}\widetilde{D}_{ii}Q_{ii}^T,$$

where $\widetilde{D}_{ii}$ is diagonal and we have

$$Q = \mathrm{diag}(Q_{11}, Q_{22}, \ldots, Q_{pp})$$

where Q_{11} is either 1 or a 2-by-2 block corresponding to the dimension of the D_{ii}, and so we have finally

$$A = LQ\overline{D}Q^T L^T, \tag{1.5.6}$$

where $\overline{D}$ is diagonal, containing the 1-by-1 D_{ii} or the 2-by-2 $\widetilde{D}_{ii}$.

See Appendix C for `lqdqtlt.m`, a MATLAB implementation.

1.5.6 The Cholesky Factorization

We look at this factorization in detail in Chapter 2.

1.6 The BLAS and LAPACK

1.6.1 Blocked Algorithms and the BLAS

All modern computers operate a *memory hierarchy*, with the fastest to access being registers where the actual computation is done down to the disk which is the slowest to access. From fastest to slowest we have

$$\text{registers} \rightarrow \text{cache} \rightarrow \text{memory} \rightarrow \text{disk}.$$

Data must move through the levels of memory to be used in the registers. Moving between levels at the bottom of the hierarchy can be slower than the arithmetic in the registers. Thus our algorithms must be designed to exploit this structure, minimizing the amount of data movement. We would like to reuse data in the cache as much as possible.

The Basic Linear Algebra Subprograms (BLAS) [32], [20], [18] are Fortran and C library routines for carrying out common linear algebra operations, and optimized versions are available for specific computer architectures. They are organized into three levels.

- Level 1 - Vector operations, such as inner products and scalar/vector multiplication.
- Level 2 - Matrix-Vector operations, such as solving triangular systems and matrix/vector multiplication.
- Level 3 - Matrix-Matrix operations, such as solving triangular systems with multiple right-hand sides and matrix/matrix multiplication.

Thus we would like to organize our algorithms so we can call these optimized subroutines. In fact, it is highly desirable that our algorithms exploit the Level 3 BLAS. Consider the three related operations

$$y = \alpha x + y, \quad y = Ax + y, \quad C = AB + C,$$

where $A, B, C \in \mathbb{R}^{n \times n}$, $x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$. Table 1.6.1 shows the amount of flops and memory references required for these operations.

Table 1.6.1: Level 1, 2 and 3 BLAS operations and the flops and memory references required.

Operation	BLAS Level	flops (f)	memory references (m)	f/m
$y = \alpha x + y$	Level 1	$2n$	$3n + 1$	$2/3$
$y = Ax + y$	Level 2	$2n^2$	$n^2 + 3n$	2
$C = AB + C$	Level 3	$2n^3$	$4n^2$	$n/2$

It is clear that Level 3 operations can have a much higher ratio of flops to memory references than lower level BLAS operations, thus reducing the data

movement overhead. This is illustrated in Figure 1.6.1 which shows the speed of the three operations in Table 1.6.1 on an IBM RS6000/590.

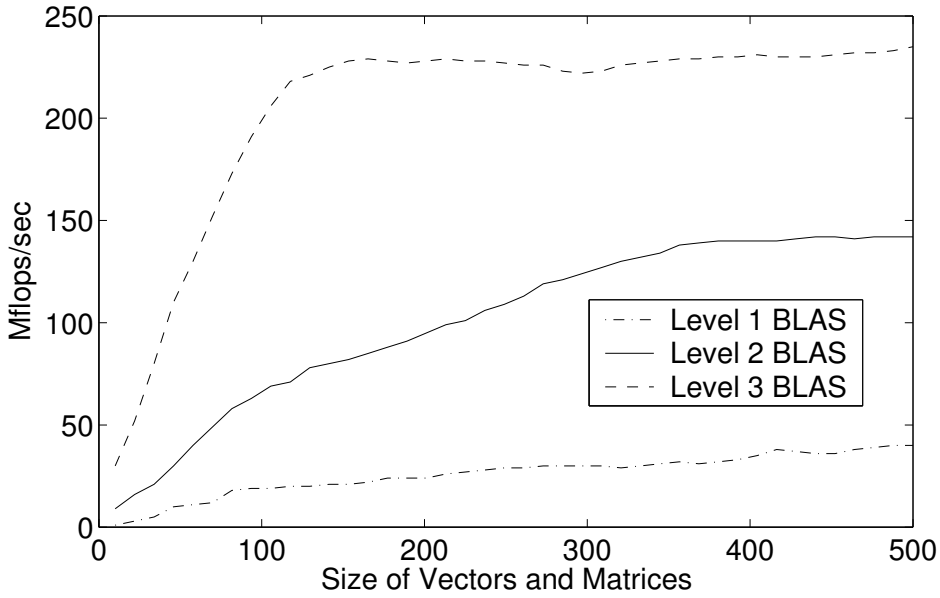

Figure 1.6.1: BLAS flop rate on IBM RS6000/590.

The Level 3 operation peaks at nearly 250 Mflops/sec. The maximum theoretical machine speed is 264 Mflops/sec. The Level 2 operation peaks at nearly 150 Mflops/sec and the Level 1 operation below 50 Mflops/sec. The plot shows the common phenomenon of speed increase for larger matrices.

We seek to write blocked algorithms like the blocked QR factorization in Section 1.4.2. Algorithm 1.4.6 involves more Level 3 BLAS operations and less Level 2 BLAS operations than the unblocked version, Algorithm 1.4.4, but does require more flops. In this case this was achieved by delaying the update of the trailing matrix until we had factored a whole *column block*. Thus by Figure 1.6.1 we would expect to achieve more Mflops/sec and an overall shorter computation time for the factorization, as the additional flops are negligible compared to the speedup the plot implies.

Also in blocked algorithms a whole block may be held in the cache memory while it is used, negating the need to move the data from main memory and potentially decreasing computation time.

The block size (n_b in Algorithm 1.4.6) must still be decided. The choice is dependent on the individual algorithm and the computer hardware it is to run on, but is typically 32 or 64.

1.6.2 LAPACK

LAPACK [1] is a library of Fortran 77 routines for common problems in numerical linear algebra. They include routines for solving linear systems of equations, least squares problem, eigenvalue problems and matrix factorizations.

The library is built on the BLAS routines as much as possible, helping to make in portable. In particular the Level 3 BLAS are exploited as much as possible.

The codes we present in Chapter 2 and Chapter 4 are written in the style of LAPACK.

1.6.3 The Choice of Block Size in our Codes.

The code in this thesis is written to fit into the LAPACK library. Thus the block size is intended to be returned by the LAPACK routine `ILAENV`, which performs this task. It is assumed that an installer of LAPACK will have sufficient knowledge of the system to amend the values in `ILAENV` where necessary.

If a user wishes to use the routines in this thesis and does not have LAPACK installed on their system, then they can remove the call to `ILAENV` and specify the block size explicitly in the source code of the routine. If the user is experienced in using the BLAS on their system than the choice of block size

may be obvious. However, if this is not the case the user is advised to start with a value of 32 and reach an optimum figure for their system empirically.

1.7 The Computing Environment

The numerical experiments reported in this thesis were run on various PCs running the Linux operating system. The details of the particular machines are given in the relevant sections.

The compiler used for the Fortran code was the GNU Fortran 77 compiler, g77, and was at version 3.2. No optimization was applied at compile time, which is the default for this compiler. Two flags were used, namely

- `Wall` which turns on all warnings the compiler can give, and
- `ffortran-bounds-check` which checks that the code does not attempt to reference an index of an array outside the specified bounds.

Thus, compilation of a code called `code.f` looked like:

```
g77 -c -Wall -ffortran-bounds-check code.f.
```

The BLAS used on the PCs was the Automatically Tuned Linear Algebra Software (ATLAS) BLAS [3]. This was compiled on a 1.4GHz AMD Athlon with 256MB of memory.

When timing of code was undertaken, the code was run 3 or 4 times to ensure consistency. The mean time is given. The PCs being used are available for remote access, thus we need to check no other user has logged in during a run of experiments, using computer resources and potentially slowing down our results. Also, since the operating system is always performing a task then a mean time will give a better estimate of the time required to run the code.

A task that requires a lot of computer resources may be carried out during one particular run giving an inaccurate estimate of time.

There is no significance in repeating the runs 3 or 4 times. This was an inconsistency of the author's over the time this thesis was written.

1.8 Performance Profiles

A convenient way of comparing the results of different methods in computation is *performance profiles*, due to Dolan and Moré [17].

Suppose we have a set of test problems, P, and a set of *solvers*, S. We distinguish solvers from methods: a solver is an implementation of a method and may not be unique. Let $t_s(p) \in \mathbb{R}$ measure the performance of solver $s \in S$ on test problem $p \in P$. This scalar measure could be typically speed, accuracy or the flop count, and a smaller measure represents a better performance. Define the performance ratio

$$r_{p,s} = \frac{t_s(p)}{\min\{t_\sigma(p) : \sigma \in S\}} \geq 1,$$

which is the performance of solver s on the problem p divided by the best performance across all solvers on this problem. The performance profile of solver s is

$$\phi_s(\theta) = \frac{1}{\text{number of problems}} \times \text{ number of } p \in P \text{ such that } r_{p,s} \leq \theta, \quad \theta \geq 1.$$

That is $\phi_s(\theta)$ is the probability that solver s is within a factor θ of the best solver on the set of test problems.

The performance profile is then generated by plotting $\phi_s(\theta)$ against θ over all solvers s. This is implemented in the MATLAB M-file `perfprof.m` [26].

Chapter 2

The Pivoted Cholesky Factorization

2.1 Introduction

The Cholesky factorization of a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$ has the form

$$A = LL^T,$$

where $L \in \mathbb{R}^{n\times n}$ is a lower triangular matrix with positive diagonal elements. If A is positive *semi*definite, of rank r, there exists a Cholesky factorization with *complete pivoting* ([29, Thm. 10.9], for example). That is, there exists a permutation matrix $P \in \mathbb{R}^{n\times n}$ such that

$$P^T AP = LL^T, \tag{2.1.1}$$

where L is unique in the form

$$L = \begin{bmatrix} L_{11} & 0 \\ L_{12} & 0 \end{bmatrix},$$

with $L_{11} \in \mathbb{R}^{r\times r}$ lower triangular with positive diagonal elements. L satisfies

$$\ell_{kk}^2 \geq \sum_{j=k}^{\min(i,r)} \ell_{ij}^2, \quad i = k+1\colon n, \quad k = 1\colon r,$$

and

$$\ell_{11} \geq \ell_{22} \geq \cdots \geq \ell_{rr}.$$

A common occurrence of positive semidefinite matrices is in statistics, namely covariance matrices. The (i,j)th element of a covariance matrix, S, holds the sample covariance of the ith and jth random variable.

If $P \in \mathbb{R}^{m\times n}$ holds m observations of n random variables then the sample covariance is defined as

$$s_{ij} = \frac{1}{m-1}(p_i - \bar{p}_i)^T(p_j - \bar{p}_j), \quad \bar{p}_k = \frac{1}{m}\sum_{\ell=1}^{m} p_{\ell k},$$

where p_k is the kth column of P and $\bar{p}_k$ is the sample mean of the kth random variable. It can be shown, see [33] for example, that S is positive semidefinite of rank at most $m-1$. Thus the number of observations need to be greater than the number of random variables for the covariance matrix to be positive definite. This is often not the case.

Covariance matrices are found in many applications such as statistical signal processing and financial modeling. The factorization of these covariance matrices arise in algorithms in theses areas, and the pivoted Cholesky factorization is therefore employed.

The factorization is also used in some algorithms solving the linear least squares problem

$$\min_x \|Ax - b\|_2, \quad A \geq 0,$$

and as a test for whether a matrix is numerically positive semidefinite or not.

However, our motivation for the work in this chapter is the efficient factorization of B in the symmetric semidefinite generalized eigenvalue problem

$$Ax = \lambda Bx, \quad B \geq 0,$$

where A and B are symmetric. This is the topic of the next chapter.

In LINPACK the routines `xCHDC` perform the Cholesky factorization with complete pivoting, but it uses only Level 1 BLAS. For computational efficiency we would like a routine that exploits the Level 2 or Level 3 BLAS.

In this chapter we describe a 'gaxpy' Level 2 BLAS algorithm for the positive definite case, and show how complete pivoting can be incorporated for the semidefinite case. We describe the existing LAPACK Level 3 code and explain why this code cannot be altered to include pivoting. We give an alternative Level 3 algorithm and show that this can include pivoting. The Level 3 code calls the Level 2 code for small n. Finally we report on some numerical experiments.

2.2 A Level 2 Gaxpy Algorithm

Gaxpy stands for *generalized Ax plus y*, and is thus a matrix-vector multiplication with a vector addition. Comparing the jth columns in $A = LL^T$ we have [24]

$$A(:,j) = \sum_{k=1}^{j} L(:,k)L^T(k,j) = \sum_{k=1}^{j} L(j,k)L(:,k),$$

and therefore

$$L(j,j)L(:,j) = A(:,j) - \sum_{k=1}^{j-1} L(j,k)L(:,k).$$

Defining

$$v := A(:,j) - \sum_{k=1}^{j-1} L(j,k)L(:,k),$$

then if we know the first $j-1$ columns of L then v is computable.

Now, $L(1\colon j-1, j) = 0$ which implies $v(1\colon j-1) = 0$ and comparing terms we have that

$$L(j,j)^2 = v(j),$$

so we have finally

$$L(j\colon n, j) = v(j\colon n)/\sqrt{v(j)},$$

which leads to the following gaxpy-based algorithm.

Algorithm 2.2.1 *This algorithm computes the Cholesky factorization $A = LL^T$ of a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$, overwriting A with L.*

```
    Set L = lower triangular part of A
    for j = 1:n
(*)     L(j,j) = L(j,j) - L(j,1:j-1)L(j,1:j-1)^T
        if L(j,j) ≤ 0
            return    % A is not positive definite
        end
        L(j,j) = √L(j,j)
        % Update jth column
        if 1 < j < n
            L(j+1:n,j) = L(j+1:n,j) - L(j+1:n,1:j-1)L(j,1:j-1)^T
        end
        if j < n
            L(j+1:n,j) = L(j+1:n,j)/L(j,j)
        end
    end
```

The (Level 2 BLAS) LAPACK [1] subroutines `xPOTF2` use this algorithm. It requires $n^3/3$ flops.

2.3 A Level 2 Pivoted Gaxpy Algorithm

We can introduce pivoting into Algorithm 2.2.1, for $L = (\ell_{ij})$, by finding the largest possible ℓ_{jj} at $(*)$ from the remaining $n-j+1$ diagonal elements and

using it as the pivot. We find

$$q = \min\Big\{p : L(p,p) - d(p) = \max_{j \le i \le n} \{L(i,i) - d(i)\}\Big\}, \tag{2.3.1}$$

where d is a vector of dot products with

$$d(i) = L(i, 1\colon j-1)L(i, 1\colon j-1)^T, \quad i = j\colon n, \tag{2.3.2}$$

and swap rows and columns q and j, putting the pivot ℓ_{qq} into the lead position. This is *complete pivoting*.

For computational efficiency we can store the inner products in (2.3.2) and update them on each iteration. This approach gives the following pivoted gaxpy algorithm.

Algorithm 2.3.1 *This algorithm computes the pivoted Cholesky factorization with complete pivoting $P^TAP = LL^T$ of a symmetric positive semidefinite matrix $A \in \mathbb{R}^{n\times n}$, overwriting A with L. The nonzero elements of the permutation matrix P are given by $P(piv(k), k) = 1$, $k = 1\colon n$.*

```
Set L = lower triangular part of A
dots(1:n) = 0        % Store accumulated dot products
piv = 1:n
for j = 1:n
    if j > 1
        dots(i) = dots(i) + L(i,j-1)^2,    i = j:n
    end
    q = min{ p : L(p,p) - dots(p) = max_{j<=i<=n} {L(i,i) - dots(i)} }
(#) if stopping criterion is met
        return      % computed rank of A is j-1
    end
    swap L(j,:) and L(q,:)
    swap L(:,j) and L(:,q)
    swap dots(j) and dots(q)
    swap piv(j) and piv(q)
    L(j,j) = L(j,j) - dots(j)
    L(j,j) = sqrt(L(j,j))
    % Update jth column
```

```
        if 1 < j < n
            L(j+1:n, j) = L(j+1:n, j) - L(j+1:n, 1:j-1)L(j, 1:j-1)^T
        end
        if j < n
            L(j+1:n, j) = L(j+1:n, j)/L(j, j)
        end
    end
```

The pivoting overhead is $3(r+1)n - 3/2(r+1)^2$ flops and $(r+1)n - (r+1)^2/2$ comparisons, where $r = \mathrm{rank}(A)$.

The computed rank, $\hat{r}$, of A is determined by a stopping criterion at (#) in Algorithm 2.3.1. For a positive semidefinite matrix A, in exact arithmetic [24, Thm. 4.2.6]

$$a_{ii} = 0 \Rightarrow A(i,:) = 0,\ A(:,i) = 0.$$

Then at the jth iteration if the pivot, which we will denote by $\chi_{jj}^{(j)}$, is less than or equal to some tolerance, tol, then

$$tol \geq \chi_{jj}^{(j)} \geq \chi_{ii}^{(j)},\ i = j+1:n,$$

and we set the trailing matrix $L(j:n, j:n) = 0$ and the computed rank is $j-1$.

Three possible stopping criteria are discussed in [29, Sec. 10.3.2]. The first is used in LINPACK's code for the Cholesky factorization with complete pivoting, `xCHDC`. Here the algorithm is stopped on the kth step if

$$\chi_{ii}^{(k)} \leq 0, \quad i = k:n. \tag{2.3.3}$$

In practice $\hat{r}$ may be greater than r due to rounding errors.

In [29] the other two criteria are shown to work more effectively. The first is

$$\|\widetilde{S}_k\| \leq \epsilon \|A\| \quad \text{or} \quad \chi_{ii}^{(k)} \leq 0, \quad i = k:n, \tag{2.3.4}$$

where $\widetilde{S}_k = A_{22} - A_{12}^T A_{11}^{-1} A_{12}$, with $A_{11} \in \mathbb{R}^{k \times k}$ the leading submatrix of A, is the Schur complement of A_{11} in A, while the second is

$$\max_{k \le i \le n} \chi_{ii}^{(k)} \le \epsilon \chi_{11}^{(1)}, \tag{2.3.5}$$

where in both cases $\epsilon = nu$, and u is the unit roundoff. The latter test is related to (2.3.4) in that if A and $\widetilde{S}_k$ are positive semidefinite then

$$\chi_{11}^{(1)} = \max_{i,j} |a_{ij}| \approx \|A\|_2, \quad \text{and,} \quad \max_{k \le i \le n} \chi_{ii}^{(k)} \approx \|\widetilde{S}_k\|_2.$$

We have used the latter criterion, preferred for its lower computational cost. See Appendix A for the double precision Fortran 77 code `lev2pchol.f`.

2.4 LAPACK's Level 3 Algorithm

The (Level 3 BLAS) LAPACK subroutines `xPOTRF` compute the Cholesky factorization of a block partitioned matrix. Starting with $L^{(0)} = A$, the algorithm computes the current block column of L using the previously computed blocks and does not update the trailing matrix. We have at the kth step

$$\begin{bmatrix} L_{11}^{(k-1)} & L_{12}^{(0)} & L_{13}^{(0)} \\ L_{21}^{(k-1)} & L_{22}^{(0)} & L_{23}^{(0)} \\ L_{31}^{(k-1)} & L_{32}^{(0)} & L_{33}^{(0)} \end{bmatrix},$$

where $L_{11}^{(k-1)} \in \mathbb{R}^{((k-1)n_b) \times ((k-1)n_b)}$, for some block size n_b, is lower triangular and we wish to update the kth block column

$$\begin{bmatrix} L_{12}^{(0)} \\ L_{22}^{(0)} \\ L_{32}^{(0)} \end{bmatrix},$$

making $L_{22}^{(k)} \in \mathbb{R}^{n_b \times n_b}$ lower triangular and $L_{12}^{(k)}$ zero. The kth step is as follows:

set $L_{12}^{(k)} = 0$
$L_{22}^{(k)} = L_{22}^{(0)} - L_{21}^{(k-1)} L_{21}^{(k-1)^T}$
factorize $L_{22}^{(k)} = \widetilde{L}\widetilde{L}^T$
if this factorization fails
    A is not positive definite
else
    $L_{22}^{(k)} = \widetilde{L}$
    $L_{32}^{(k)} = L_{32}^{(0)} - L_{31}^{(k-1)} L_{21}^{(k-1)^T}$
    solve $X L_{22}^{(k)} = L_{32}^{(k)}$, for X
    $L_{32}^{(k)} = X$
end

In order to add pivoting to this algorithm we would need to decide all the pivots for the kth block column, carry out the required permutations, and continue with the step above.

The pivot for the first column can be found by computing all the possible diagonal elements as (2.3.1). To repeat this to find the second pivot we need first to update the vector of dot products, which can only be achieved by updating the first column of the kth block. So we have performed a complete step of Algorithm 2.3.1, before we find the second pivot. Thus in determining all the pivots for the current block column we will have formed $L_{22}^{(k)}$ and $L_{32}^{(k)}$ by Algorithm 2.3.1. We would like an algorithm with Level 3 operations that is independent of the pivoting.

2.5 A Level 3 Pivoted Algorithm

We can write for the semidefinite matrix $A^{(k-1)} \in \mathbb{R}^{n \times n}$ and $n_b \in \mathbb{R}$ [24]

$$A^{(k-1)} = \begin{bmatrix} A_{11}^{(k-1)} & A_{12}^{(k-1)} \\ A_{12}^{T(k-1)} & A_{22}^{(k-1)} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-n_b} \end{bmatrix} \begin{bmatrix} I_{n_b} & 0 \\ 0 & A^{(k)} \end{bmatrix} \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-n_b} \end{bmatrix}^T,$$

where $L_{11} \in \mathbb{R}^{n_b \times n_b}$ and $L_{21} \in \mathbb{R}^{(n-n_b) \times n_b}$ form the first n_b columns of the Cholesky factor L of $A^{(k-1)}$. Now to complete our factorization of $A^{(k-1)}$ we

need to factor the reduced matrix

$$A^{(k)} = A_{22}^{(k-1)} - L_{21}L_{21}^T, \tag{2.5.1}$$

which we can explicitly form, taking advantage of symmetry.

From this representation we can derive a block algorithm. We start with $A^{(0)} = A$ and a block size n_b. At the kth step we apply the equivalent of n_b steps of Algorithm 2.3.1 to $A^{(k-1)}$ to form n_b columns of L. We then update the trailing matrix $A^{(k)}$, which is of dimension $n-kn_b$, according to (2.5.1). We then repeat the process with $A^{(k)}$, the factorization of which is independent of the columns of L already formed. This is a pivoted algorithm, as Algorithm 2.3.1 acts on the whole trailing matrix.

At each step the Level 2 part of the algorithm requires $(n-(k-1)n_b)n_b^2$ flops and the Level 3 update requires $(n-kn_b)^3/3$ flops. The Level 3 fraction is approximately $1-3n_b/2n$.

Algorithm 2.5.1 *This algorithm computes the pivoted Cholesky factorization with complete pivoting $P^TAP = LL^T$ of a symmetric positive semidefinite matrix $A \in \mathbb{R}^{n\times n}$, overwriting A with L, using a Level 3 update and block size n_b. The nonzero elements of the permutation matrix P are given by $P(piv(k),k) = 1$, $k = 1\colon n$.*

Set L = lower triangular part of A
$\epsilon = nu$
$piv = 1\colon n$
for $k = 1\colon n_b\colon n$
    $jb = \min(n_b, n-k+1)$ % Allow for last incomplete block
    $dots(k\colon n) = 0$ % Store accumulated dot products
    $tol = n * u * \max(\mathrm{diag}(A))$ % Tolerance in stopping criterion
    for $j = k\colon k+jb-1$
        if $j > k$
            $dots(i) = dots(i) + L(i,j-1)^2, \quad i = j\colon n$
        end
        $q = \min\Big\{p : L(p,p) - dots(p) = \max_{j\le i\le n}\{L(i,i) - dots(i)\}\Big\}$

```
            if L(q,q) ≤ tol
                return    % computed rank of A is j − 1
            end
            swap L(j,:) and L(q,:)
            swap L(:,j) and L(:,q)
            swap dots(j) and dots(q)
            swap piv(j) and piv(q)
            L(j,j) = L(j,j) − dots(j)
            L(j,j) = √L(j,j)
            % Update jth column
            if 1 < j < n
                L(j+1:n,j) = L(j+1:n,j)−
                                L(j+1:n,1:j−1)L(j,1:j−1)^T
            end
            if j < n
                L(j+1:n,j) = L(j+1:n,j)/L(j,j)
            end
        end
        if k + jb < n
        % perform Level 3 update
            L(j+1:n,j+1:n) = L(j+1:n,j+1:n)−
                                L(j+1:n,1:j)L(j+1:n,1:j)^T
        end
    end
```

See Appendix A for the double precision Fortran 77 code `lev3pchol.f`.

2.6 Numerical Experiments

We tested and compared four Fortran subroutines:

- LINPACK's `DCHDC` [21], which uses Level 1 BLAS and stopping criterion (2.3.3).
- LINPACK's `DCHDC`, altered to use stopping criterion (2.3.5).

- An implementation of Algorithm 2.3.1, obtained by modifying LAPACK's `DPOTF2`, using stopping criterion (2.3.5): `lev2pchol.f` in Appendix A.

- An implementation of Algorithm 2.5.1, again using stopping criterion (2.3.5): `lev3pchol.f` in Appendix A

The tests were performed on a 1400MHz AMD Athlon running Red Hat Linux version 6.2 with kernel 2.2.22 and a Sun 167MHz UltraSparc running Solaris version 2.7. All test matrices were generated in MATLAB 6.5. The unit roundoff $u \approx$ 1.1e-16.

We test the speed of computation, the normwise backward error and the rank revealing properties of the routines. In all cases the results for the Linux machine are given. The results on the Sun for backward error and rank detection were indistinguishable from those on the Linux machine. The timings on the Sun were much greater than, but in proportion to, those on the Linux machine.

2.6.1 Speed Tests

We first compared the speed of the factorization of the LINPACK code and our Level 2 and 3 routines for different sizes of $A \in \mathbb{R}^{n\times n}$. We generated random symmetric positive semidefinite matrices of order n and rank $r = 0.7n$ by computing

$$A = \sum_{i=1}^{0.7n} xx^T, \quad x = \mathrm{rand}(n,1),$$

where the MATLAB command $\mathrm{rand}(n,1)$ generates a random vector, with elements uniformly distributed on $(0,1)$, of length n. For each value of n the codes were run four times and the mean times are shown in Figure 2.6.1. The speedups of the new codes over the LINPACK code are given in Table 2.6.1.

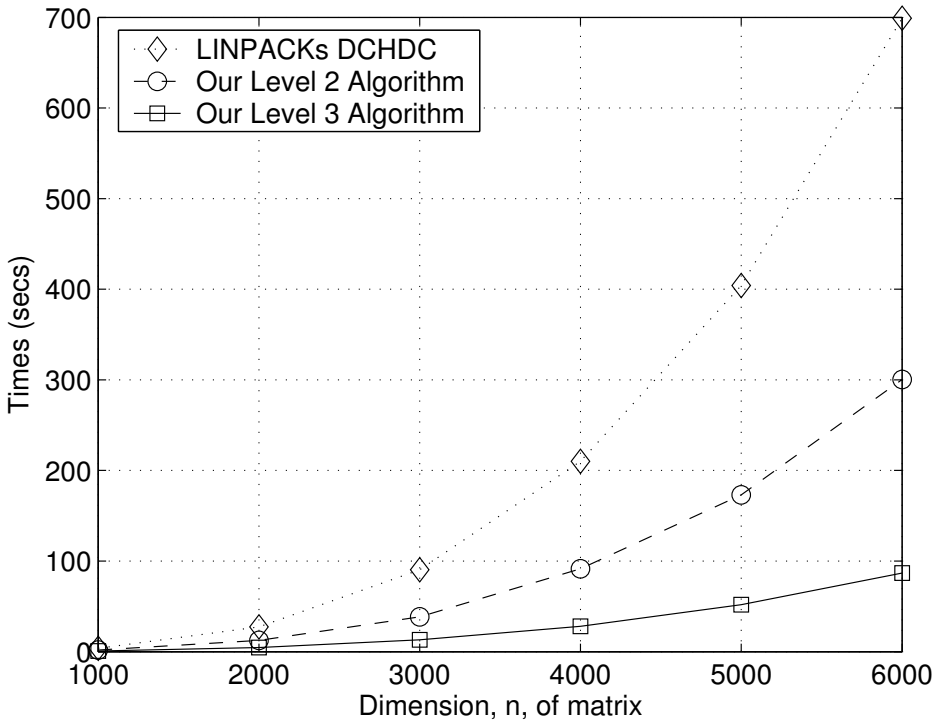

Figure 2.6.1: Comparison of speed for different n.

We achieve a good speedup, with the Level 3 code as much as 8 times faster than the LINPACK code.

We also compared the speed of the unpivoted LAPACK subroutines against our Level 2 and Level 3 pivoted codes, using full rank matrices, to demonstrate the pivoting overhead. Figure 2.6.2 shows the ratio of speed of the pivoted codes to the unpivoted codes. These show that the pivoting overhead is negligible for large n, recalling the pivoting overhead is $3rn - 3/2r^2$ flops, and the unpivoted factorization is of order n^3. The use of the pivoted codes instead of the unpivoted ones could be warranted if there is any doubt over whether a matrix is positive definite.

Table 2.6.1: Speedups of our codes compared with LINPACK code.

n	1000	2000	3000	4000	5000	6000
`LEV2PCHOL`	2.05	2.21	2.35	2.29	2.33	2.32
`LEV3PCHOL`	5.30	6.03	6.90	7.52	7.78	8.05

2.6.2 Backward Error Tests and Rank Detection

We tested all four subroutines on a further set of random positive semidefinite matrices, this time with pre-determined eigenvalues, similar to the tests in [27]. The matrices were generated by setting $A = Q\Lambda Q^T$ where Q was a random orthogonal matrix computed by the method of Stewart [48] using `qmult.m` [35]. For matrices of rank r we chose the nonzero eigenvalues in three ways:

- Case 1: $\lambda_1 = \lambda_2 = \cdots = \lambda_{r-1} = 1, \quad \lambda_r = \alpha \leq 1$
- Case 2: $\lambda_1 = 1, \quad \lambda_2 = \lambda_3 = \cdots = \lambda_r = \alpha \leq 1$
- Case 3: $\lambda_i = \alpha^{i-1}, \quad 1 \leq i \leq r, \quad \alpha \leq 1$

Here, α was chosen to vary $\kappa_2(A) = \lambda_1/\lambda_r$.

For each case we constructed a set of 100 matrices by using every combination of:

$$\begin{aligned} n &= \{70,\ 100,\ 200,\ 500,\ 1000\}, \\ \kappa_2(A) &= \{1,\ 1e{+}3,\ 1e{+}6,\ 1e{+}9,\ 1e{+}12\}, \\ r &= \{0.2n,\ 0.3n,\ 0.5n,\ 0.9n\}, \end{aligned}$$

where $r = \mathrm{rank}(A)$. We computed the relative normwise backward error

$$\frac{\|A - \widehat{P}\widehat{L}\widehat{L}^T\widehat{P}^T\|_2}{\|A\|_2},$$

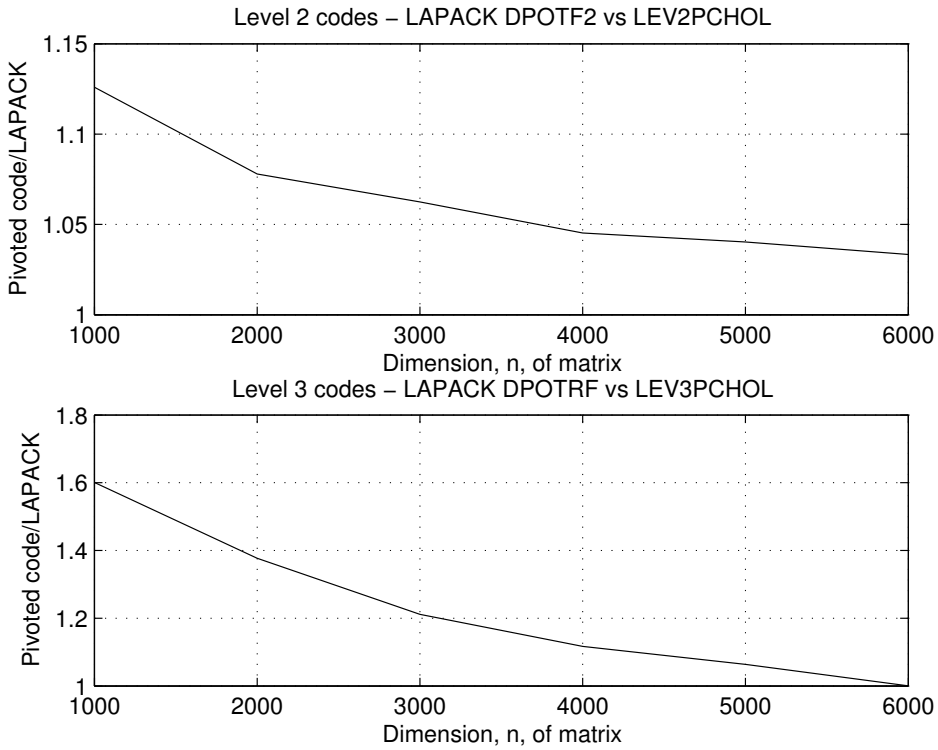

Figure 2.6.2: Speed ratio of pivoted codes over LAPACK codes for the full rank case.

for the computed Cholesky factor $\widehat{L}$ and permutation matrix $\widehat{P}$. We have, from [29, Thm. 10.22], the following upper bound

$$\frac{\|A - \widehat{P}\widehat{L}\widehat{L}^T\widehat{P}^T\|_2}{\|A\|_2} \leq 2r\gamma_{r+1}(\|W\|_2 + 1)^2 + O(u^2), \tag{2.6.1}$$

where

$$W = \widehat{L}_{11}^{-1}\widehat{L}_{12}, \quad \widehat{L} = \begin{bmatrix} \widehat{L}_{11} & 0 \\ \widehat{L}_{12} & 0 \end{bmatrix}, \quad \widehat{L}_{11} \in \mathbb{R}^{r\times r}, \quad \gamma_{r+1} = \frac{c(r+1)u}{1 - c(r+1)u},$$

and u is the unit roundoff, c is a small integer constant, and from [29, Lemma 10.13],

$$\|W\|_2 \leq \sqrt{\frac{1}{3}(n-r)(4^r - 1)}, \tag{2.6.2}$$

and so there is no guarantee of stability of the algorithm for large n and r.

Table 2.6.2: Comparison of normwise backward errors.

n		70	100	200	500	1000
$\|W\|_2$	min	3.58	4.39	7.91	15.12	27.11
	max	10.67	12.26	20.62	32.52	66.03
DCHDC	min	1.654e-16	3.654e-16	6.651e- 16	5.504e-15	1.933e-14
	max	3.172e-13	1.498e-13	1.031e-12	2.823e-12	4.737e-11
DCHDC	min	1.707e-16	2.561e-16	4.737e-16	1.273e-15	2.687e-15
with (2.3.5)	max	7.778e-15	9.014e-15	1.810e-14	7.746e-14	1.991e-13
LEV2PCHOL	min	1.671e-16	2.526e-16	5.121e-16	1.240e-15	2.597e-15
	max	4.633e-15	9.283e-15	1.458e-14	7.290e-14	1.983e-13
LEV3PCHOL	min	1.671e-16	2.476e-16	5.121e-16	1.271e-15	2.600e-15
	max	4.633e-15	9.283e-15	1.710e-14	8.247e-14	2.049e-13

There was little difference for the normwise backward errors between the three cases and Table 2.6.2 shows minimum and maximum values for different n. The codes with the new stopping criterion give smaller errors than the original LINPACK code. The minimum and maximum values of $\|W\|$ are also given, and show that after r stages the algorithm will have produced a stable factorization. In fact, for all the codes with our stopping criterion $\hat{r} = r$, and so the rank was detected exactly. This was not the case for the unmodified DCHDC, and the error, $\hat{r} - r$, is shown in Table 2.6.3.

Table 2.6.3: Errors in computed rank for DCHDC.

n	70	100	200	500	1000
min	0	0	1	4	4
max	10	12	16	16	19

The larger backward error for the original DCHDC is due to the stopping criterion. As Table 2.6.3 shows, the routine is terminated after more steps than our codes, adding more nonzero columns to $\widehat{L}$.

2.7 Checking for Indefiniteness

The algorithm does not attempt to check if the matrix is positive semidefinite and indeed the stopping criterion is based on A being positive semidefinite. For example, if we supplied

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

then the algorithm would stop after one step, and give the rank to be 1, whereas A is indefinite with eigenvalues of $\{1, 1, -1\}$. A check of the residual

$$\|A - \widehat{P}\widehat{L}\widehat{L}^T\widehat{P}^T\|,$$

bearing in mind (2.6.1) and (2.6.2), may allow confirmation that A is not close to being positive semidefinite.

The user could also update the trailing matrix by (2.5.1) to give

$$\widetilde{A}^{r+1} = A^{(r+1)}(r+1{:}\,n, r+1{:}\,n) = A^{(r)}(r+1{:}\,n, r+1{:}\,n) - \widehat{L}_{21}\widehat{L}_{21}^T.$$

We know the diagonal elements should be 'small' since the algorithm stopped after $\hat{r}$ steps and consequently $\|\widetilde{A}^{r+1}\|$ should be negligible, but if $\|\widetilde{A}^{r+1}\|$ is large then the original matrix could have been indefinite. However, if $\|W\|$ is large then it is more difficult to reach this conclusion, as there may have been significant error growth during the factorization.

Of course, if there is serious doubt over semidefiniteness then a symmetric indefinite factorization should be used.

2.8 Conclusions

We have presented two Fortran 77 codes for the Cholesky factorization with complete pivoting of a positive semidefinite matrix: a Level 2 BLAS version

and a Level 3 BLAS version. Our tests show that our codes are much faster than the existing LINPACK code on our test machines.

Furthermore, with a new stopping criterion the rank is revealed much more reliably, and this leads to a smaller normwise backward error.

We propose that the double precision Fortran 77 codes `lev2pchol.f` and `lev3pchol.f`, and their single precision and complex equivalents, be included in LAPACK.

Appendix B contains Fortran 77 testing routines required for submission of the codes in Appendix A to LAPACK.

Chapter 3

The Symmetric Semidefinite Generalized Eigenvalue Problem

3.1 Introduction

We consider the generalized eigenvalue problem

$$Ax = \lambda Bx \tag{3.1.1}$$

in the case where A and B are real and symmetric and B is positive semidefinite. However, our theory and algorithms extend to the complex Hermitian case.

The symmetric semidefinite generalized eigenvalue problem arises in structural engineering and commonly in vibrational analysis. A vibration analysis requires the solution of the problem

$$Kx = \lambda Mx,$$

where K is the stiffness matrix and M is the mass matrix and is positive semidefinite. The matrices are derived from a finite element analysis of the vibration problem. The λ are the natural frequencies and the x the normal

mode shapes. The solution of the generalized eigenvalue problem form the solution of the ordinary differential equation

$$M\ddot{y}(t) + Ky(t) = 0.$$

By considering solutions of the form

$$y(t) = xe^{i\omega t},$$

where x is a nonzero vector we have the generalized eigenvalue problem

$$-\omega^2 e^{i\omega t}Mx + e^{i\omega t}Kx = -\omega^2 Mx + Kx = 0, \quad -\omega^2 = \lambda,$$

as $e^{i\omega t}$ is nonzero.

Since the matrices are from a finite element analysis they can be arbitrarily large. Dimensions of M and K in the tens of thousand are common.

The symmetric semidefinite generalized eigenvalue problem is used in another structural engineering problem, including buckling analysis. They also arise in quantum chemistry, electronic engineering and control theory.

Because the dimensions of the matrices can be very large we would like an algorithm that is efficient as possible and exploits the symmetry of the problem.

We show how to reduce (3.1.1) by congruence transformations to a symmetric standard eigenvalue problem. If the pencil $A - \lambda B$ is nonregular we deflate it to a regular pencil during the reduction. Our algorithm extends the work of Fix and Heiberger [23], as generalized by Parlett [41] and Cao [11]. We generalize the algorithm further by considering a wider choice of congruence transformations by not limiting ourselves to orthogonal transformations.

We will see that our choice of transformations leads to an algorithm that has many times less flops than those suggested by the authors above. Also, the backward error of the computed solution can be as favourable as as that

computed with orthogonal transformations. Thus we show it is possible to increase efficiency of existing algorithms without the lost of numerical stability.

We also compare our algorithm with other methods, neither of which exploit the symmetry of the problem. Namely the QZ algorithm for regular matrix pencils and the GUPTRI algorithm for nonregular pencils. We show that the QZ algorithm produces a solution with a smaller error than is produced by implementations of our algorithm but requires far more flops. Also the QZ algorithm is unsuitable for nonregular pencils. The GUPTRI algorithm that can deal with nonregular pencils always has a larger error than our algorithm in our experiments. Also the implementation of our algorithm with nonorthogonal transformations produces errors comparable with those produced by orthogonal transformations. The flops required by the GUPTRI algorithm is an order of magnitude higher than that required by ours. The GUPTRI algorithm also fails for some of our test problems.

Thus in the general case of nonregular matrix pencils we propose an algorithm that is more efficient and can be as stable as algorithms previously suggested.

The efficient implementation of our algorithm depends upon the algorithm and code we developed in Chapter 2.

We give a full classification of the eigenvalues of the problem and discuss the geometric multiplicity of the eigenvectors.

We will use the notation $A, B \xrightarrow{U} A', B'$ to denote the transformation

$$A' = U^T A U, \quad B' = U^T B U.$$

If $B = X D X^T$ and X is singular then we define $\overline{X}$ such that

$$A, B \xrightarrow{\overline{X}^{-T}} A', D,$$

where $\overline{X}$ is nonsingular. For example, the pivoted Cholesky factorization of P^TBP yields a factor

$$X = L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & 0 \end{bmatrix},$$

where L_{11} is nonsingular with dimension the rank of B, and X is singular. We define

$$\overline{X} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix},$$

and

$$A, P^TBP \xrightarrow{\overline{X}^{-T}} A', \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}.$$

For a general matrix, C, with a factorization, $C = XGZ$, we form transformation matrices in an equivalent way.

3.2 A General Method

3.2.1 Step One

First we diagonalize the symmetric positive semidefinite matrix B by computing an RRD (rank revealing decomposition)

$$B = X_1 \mathrm{diag}(D_1, D_2) X_1^T,$$

where $D_1 \in \mathbb{R}^{r_1 \times r_1}$ is positive definite and $D_2 \in \mathbb{R}^{r_2 \times r_2}$, with r_1 chosen minimally so that $\|D_2\|_2 \leq \eta \|D_1\|_2$ and η is described in Section 1.5. We set $D_2 = 0$. We thus transform the pencil $A - \lambda B$ according to

$$A, B \xrightarrow{\overline{X_1}^{-T}} A^{(1)}, B^{(1)} = \begin{array}{c} \\ r_1 \\ r_2 \end{array} \begin{array}{c} \begin{array}{cc} r_1 & r_2 \end{array} \\ \begin{bmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ A_{12}^{(1)T} & A_{22}^{(1)} \end{bmatrix} \end{array}, \mathrm{diag}(D_1, 0).$$

3.2.2 Step Two

We now transform D_1 to the identity. We define $U_2 = \mathrm{diag}(D_1^{-1/2}, I_{r_2})$ and then

$$A^{(1)}, B^{(1)} \xrightarrow{U_2} A^{(2)}, B^{(2)} \equiv \begin{array}{c} \\ r_1 \\ r_2 \end{array} \begin{array}{c} \begin{array}{cc} r_1 & r_2 \end{array} \\ \begin{bmatrix} A_{11}^{(2)} & A_{12}^{(2)} \\ A_{12}^{(2)T} & A_{22}^{(2)} \end{bmatrix} \end{array}, \mathrm{diag}(I_{r_1}, 0).$$

If B is positive definite then $r_2 = 0$ and we can now solve the standard eigenvalue problem

$$A^{(2)} z = \lambda z, \quad x = \overline{X}_1^{-T} U_2 z.$$

If B is positive semidefinite we proceed to step three.

3.2.3 Step Three

We now reduce $A_{22}^{(2)}$ to diagonal form. Compute the RRD

$$A_{22}^{(2)} = X_3 \Phi_3 X_3^T,$$

where the diagonal matrix

$$\Phi_3 = \begin{array}{c} \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{cc} r_3 & r_4 \end{array} \\ \begin{bmatrix} \Phi_{11} & 0 \\ 0 & \Phi_{22} \end{bmatrix} \end{array},$$

where $r_2 = r_3 + r_4$ and r_3 is chosen minimally so that $\|\Phi_{22}\|_2 \leq \eta \|\Phi_{11}\|_2$. We set $\Phi_{22} = 0$ and define $U_3 = \mathrm{diag}(I_{r_1}, \overline{X}_3^{-T})$ and hence transform

$$A^{(2)}, B^{(2)} \xrightarrow{U_3} A^{(3)}, B^{(3)},$$

where

$$A^{(3)} = \begin{array}{c} \\ r_1 \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{ccc} r_1 & r_3 & r_4 \end{array} \\ \begin{bmatrix} A_{11}^{(3)} & A_{12}^{(3)} & A_{13}^{(3)} \\ A_{12}^{(3)T} & D_{22}^{(3)} & 0 \\ A_{13}^{(3)T} & 0 & 0 \end{bmatrix} \end{array}, \quad B^{(3)} = B^{(2)} = \mathrm{diag}(I_{r_1}, 0),$$

and $D_{22}^{(3)} = \Phi_{11}$.

Now if $r_4 = 0$, the last row and column of $A^{(3)}$ is not present and we have the eigenproblem, which we now write in (α, β) form $(\lambda = \alpha/\beta)$,

$$\begin{array}{c} \\ r_1 \\ r_2 \end{array} \beta \begin{array}{c} \begin{array}{cc} r_1 & r_2 \end{array} \\ \begin{bmatrix} A_{11}^{(3)} & A_{12}^{(3)} \\ A_{12}^{(3)T} & D_{22}^{(3)} \end{bmatrix} \end{array} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \alpha \begin{bmatrix} I_{r_1} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}. \tag{3.2.1}$$

Solving for w_2 gives $w_2 = -D_{22}^{(3)^{-1}} A_{12}^{(3)^T} w_1$ and hence we obtain the r_1-by-r_1 symmetric eigenvalue problem

$$\beta(A_{11}^{(3)} - A_{12}^{(3)} D_{22}^{(3)^{-1}} A_{12}^{(3)^T}) w_1 = \alpha w_1. \tag{3.2.2}$$

Thus there are r_1 finite eigenvalues. There are also r_2 infinite eigenvalues corresponding to $\beta = 0$. Solving (3.2.2) in this case gives $w_1 = 0$, and the w_2 are arbitrary provided they are linearly independent, so we set r_2 $w_2 = e_i \in \mathbb{R}^{r_2}$, $i = 1\colon r_2$. The eigenvectors of the original problem are then given by

$$x = \overline{X}_1^{-T} U_2 U_3 w.$$

The eigenvectors corresponding to the infinite eigenvalues are the null vectors of B.

If $r_4 > 0$ and $A_{13}^{(3)} = 0$ then $\det(A^{(3)} - \lambda B^{(3)}) \equiv 0$ and hence the pencil $A - \lambda B$ is nonregular. On removing the last r_4 rows and columns we deflate the nonregular part of the pencil and we obtain a system of the form (3.2.1) again, which provides r_1 finite and r_3 infinite eigenvalues.

If neither $r_4 = 0$ nor $A_{13}^{(3)} = 0$ then we proceed to step four.

3.2.4 Step Four

Now we transform $A_{13}^{(3)} \in \mathbb{R}^{r_1 \times r_4}$. This matrix can be of arbitrary rank, so we compute the RRD,

$$A_{13}^{(3)} = X_4 \begin{array}{c} \\ r_5 \\ r_1 - r_5 \end{array} \begin{array}{c} \begin{array}{cc} r_5 & r_4 - r_5 \end{array} \\ \begin{bmatrix} G_{11} & 0 \\ 0 & 0 \end{bmatrix} \end{array} Z_4,$$

where G_{11} is diagonal or triangular and X_4 is required to be orthogonal to keep $B = \mathrm{diag}(I_{r_1}, 0)$ on transformation. This decomposition will involve some η-negligibility decisions during its computation. We define $U_4 = \mathrm{diag}(X_4, I_{r_3}, \overline{Z}_4^{-1})$, and perform a final transformation

$$A^{(3)}, B^{(3)} \xrightarrow{U_4} A^{(4)}, B^{(4)},$$

where

$$A^{(4)} = \left[\begin{array}{ccc|cc} A_{11}^{(4)} & A_{12}^{(4)} & A_{13}^{(4)} & G_{14}^{(4)} & 0 \\ A_{12}^{(4)^T} & A_{22}^{(4)} & A_{23}^{(4)} & 0 & 0 \\ \hline A_{13}^{(4)^T} & A_{23}^{(4)^T} & D_{33}^{(4)} & 0 & 0 \\ G_{14}^{(4)^T} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \begin{array}{l} r_5 \\ r_6 := r_1 - r_5 \\ r_3 \\ r_5 \\ r_7 := r_4 - r_5 \end{array} \begin{array}{l} \Big\} r_1 \\ \\ \Big\} r_4 \end{array}$$

and $G_{14}^{(4)} = G_{11}$, $D_{33}^{(4)} = D_{22}^{(3)}$ and $B^{(4)} = B^{(3)} = \mathrm{diag}(I_{r_1}, 0)$.

The last block row and column of zeros are not present if $r_7 = 0$, and the pencil is then regular; otherwise it is nonregular.

After removing the last block row and column of zeros, if necessary, we are left with the following regular eigenproblem, which we again write in (α, β) form,

$$\beta \begin{bmatrix} A_{11}^{(4)} & A_{12}^{(4)} & A_{13}^{(4)} & G_{14}^{(4)} \\ A_{12}^{(4)^T} & A_{22}^{(4)} & A_{23}^{(4)} & 0 \\ A_{13}^{(4)^T} & A_{23}^{(4)^T} & D_{33}^{(4)} & 0 \\ G_{14}^{(4)^T} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \alpha \begin{bmatrix} I_{r_5} & 0 & 0 & 0 \\ 0 & I_{r_6} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}. \quad (3.2.3)$$

Solving from the bottom up gives

$$\begin{aligned} \beta v_1 &= 0, \\ \beta v_3 &= -\beta D_{33}^{(4)^{-1}} (A_{13}^{(4)^T} v_1 + A_{23}^{(4)^T} v_2) \\ &= -\beta D_{33}^{(4)^{-1}} A_{23}^{(4)^T} v_2, \\ \beta (A_{22}^{(4)} - A_{23}^{(4)} D_{33}^{(4)^{-1}} A_{23}^{(4)^T}) v_2 &= \alpha v_2, \\ \beta v_4 &= G_{14}^{(4)^{-1}} (\alpha v_1 - \beta A_{12}^{(4)} v_2 - \beta A_{13}^{(4)} v_3). \end{aligned} \quad (3.2.4)$$

We distinguish two cases. First, $\beta \neq 0$: here, (3.2.4) gives r_6 finite eigenvalues, $v_1 = 0$, and v_3 and v_4 are uniquely determined by the eigenvectors v_2. Second, $\beta = 0$, and (necessarily) $\alpha \neq 0$: from (3.2.3) it is clear that $v_1 = 0$, $v_2 = 0$ and v_3 and v_4 are arbitrary. Therefore there are $r_3 + r_5$ linearly independent eigenvectors corresponding to infinite eigenvalues. Our total of $r_6 + r_3 + r_5$ eigenvalues of (3.2.3) leaves r_5 eigenvalues unaccounted for. It is not hard to show that the determinant of the pencil in (3.2.3) is

$$\pm \det(\beta G_{14}^{(4)})^2 \det(\beta D_{33}^{(4)}) \det(\beta A_{22}^{(4)} - \beta A_{23}^{(4)} D_{33}^{(4)^{-1}} A_{23}^{(4)^T} - \alpha I_{r_6}),$$

and hence there are $2r_5 + r_3$ infinite eigenvalues in total, corresponding to $\beta = 0$, but only $r_5 + r_3$ corresponding linearly independent eigenvectors, given by setting

$$\begin{bmatrix} v_3 \\ v_4 \end{bmatrix} = e_i \in \mathbb{R}^{r_3 + r_5}, \ i = 1\colon r_3 + r_5.$$

In the nonregular case we set

$$v = [\, 0 \quad v_2^T \quad v_3^T \quad v_4^T \quad 0 \,]^T,$$

and we have the corresponding eigenvectors of the original problem (3.1.1),

$$x = \overline{X}_1^{-T} U_2 U_3 U_4 v.$$

3.2.5 Summary

The classification of the eigenvalues provided by the algorithm can be summarized as follows.

if $r_4 = 0$

The problem is regular, with r_1 finite and r_2 infinite eigenvalues.

else

if $A_{13}^{(3)} = 0$

The problem is nonregular, with r_1 finite and r_3 infinite eigenvalues.

else

The problem is nonregular if $r_7 > 0$ and otherwise regular.

There are r_6 finite and $r_3 + 2r_5$ infinite eigenvalues.

end

All the finite eigenvalues are real, as they arise as the solutions of symmetric standard eigenproblems.

Also we note that if $r_1 \leq r_4$ and $A_{13}^{(3)}$ has full rank, then $r_6 = 0$ and there are no finite eigenvalues.

3.3 Existing Methods

3.3.1 Fix and Heiberger's Algorithm

Our algorithm is a generalization of the Fix and Heiberger algorithm [23]. Their algorithm uses the following RRDs:

- Step one: spectral decomposition,
- Step three: spectral decomposition,
- Step four: QR factorization.

The QR factorization of $A_{13}^{(3)}$ at step four requires the conditions

$$r_1 \geq r_4 \quad \text{and} \quad \text{rank}(A_{13}^{(3)}) = r_4,$$

and therefore is not applicable to nonregular pencils.

Some numerical examples are given in [23]. They also derive error bounds for computed eigenvalues and eigenvectors.

3.3.2 Parlett's Algorithm

Parlett's algorithm [41] improves on Fix and Heiberger by allowing for nonregular pencils. The first three steps are the same but at step four the QR factorization is replaced with the singular value decomposition

$$A_{13}^{(3)} = U \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} V^T,$$

and with $\|\Sigma_2\| \leq \eta \|\Sigma_1\|$, Σ_2 is set to zero, for some suitable η.

3.3.3 Cao's Algorithm

Cao's algorithm [11], like ours, is a generalization of Fix and Heiberger's algorithm. Here, the only requirement is that the transformations at steps one, three and four are all orthogonal.

3.3.4 The QZ Algorithm

The QZ algorithm of Moler and Stewart [37] computes the generalized Schur form (1.2.4) to solve the generalized eigenvalue problem. There are three steps.

Step One - Reduction to Hessenberg-Triangular Form

The first step reduces A to upper Hessenberg form and B to upper triangular form. An orthogonal matrix U is found such that, in the 5-by-5 case

$$U^T A = \begin{bmatrix} + & + & + & + & + \\ + & + & + & + & + \\ + & + & + & + & + \\ + & + & + & + & + \\ + & + & + & + & + \end{bmatrix}, \quad U^T B = \begin{bmatrix} + & + & + & + & + \\ 0 & + & + & + & + \\ 0 & 0 & + & + & + \\ 0 & 0 & 0 & + & + \\ 0 & 0 & 0 & 0 & + \end{bmatrix},$$

where $+$ represents a nonzero entry. Then zeros are introduced into A by Givens matrices while keeping the upper triangular structure of B. Starting with the bottom left corner A and B are multiplied by a Givens matrix from the left such that

$$A, B \leftarrow G_L^T(4,5)(A,B) = \begin{bmatrix} + & + & + & + & + \\ + & + & + & + & + \\ + & + & + & + & + \\ + & + & + & + & + \\ 0 & + & + & + & + \end{bmatrix}, \begin{bmatrix} + & + & + & + & + \\ 0 & + & + & + & + \\ 0 & 0 & + & + & + \\ 0 & 0 & 0 & + & + \\ 0 & 0 & 0 & + & + \end{bmatrix}.$$

A nonzero entry is introduced in B which is immediately eliminated by multiplying A and B with a Givens matrix from the right such that

$$A, B \leftarrow (A,B)G_R(4,5) = \begin{bmatrix} + & + & + & + & + \\ + & + & + & + & + \\ + & + & + & + & + \\ + & + & + & + & + \\ 0 & + & + & + & + \end{bmatrix}, \begin{bmatrix} + & + & + & + & + \\ 0 & + & + & + & + \\ 0 & 0 & + & + & + \\ 0 & 0 & 0 & + & + \\ 0 & 0 & 0 & 0 & + \end{bmatrix}.$$

We continue in this manner with pairs of Givens matrices applied to the pencil until we have finally

$$\begin{aligned} A - \lambda B \leftarrow\ & G_L^T(4,5)G_L^T(3,4)G_L^T(4,5)G_L^T(2,3)G_L^T(3,4)G_L^T(4,5) * \\ & (A - \lambda B)G_R(4,5)G_R(3,4)G_R(2,3)G_R(4,5)G_R(3,4)G_R(4,5), \end{aligned}$$

where A is now upper Hessenberg and B is upper triangular as required.

Step Two - Deflation

If $a_{k+1,k} = 0$, for some k, then we have

$$A - \lambda B = \begin{array}{c} \\ k \\ n-k \end{array} \begin{array}{c} \begin{array}{cc} k & \quad n-k \end{array} \\ \begin{bmatrix} A_{11} - \lambda B_{11} & A_{12} - \lambda B_{12} \\ 0 & A_{22} - \lambda B_{22} \end{bmatrix} \end{array},$$

and we can solve the two smaller problems $A_{11} - \lambda B_{11}$ and $A_{22} - \lambda B_{22}$ independently.

If a zero appears on the diagonal of B, in any position, it is possible to introduce a zero in the $(n, n-1)$ position of A and also *chase* the zero on B's diagonal to the (n, n) position, and thus deflate the problem. This is achieved with pairs of Givens matrices in a similar way as used in step one. After this process we have

$$A - \lambda B = \begin{bmatrix} A_{11} & A_{12} \\ 0 & a_{22} \end{bmatrix} - \lambda \begin{bmatrix} B_{11} & B_{12} \\ 0 & 0 \end{bmatrix},$$

where A_{11} is upper Hessenberg, $a_{22} \neq 0$ is a scalar and B_{11} is upper triangular. We thus solve $A_{11} - \lambda B_{11}$ and have an infinite eigenvalue given by

$$\lambda = \frac{a_{22}}{0}.$$

Step Three - The QZ Step

The QZ step applies unitary transformations, Q and Z to the deflated $A - \lambda B$, such that

$$Q^*(A - \lambda B)Z = S - \lambda T,$$

where S and T are upper triangular. In the real case S may only be reduced to *quasi*upper triangular form. That is block upper triangular with 1-by-1 and 2-by-2 blocks. It can be reduced to upper triangular form if S is allowed to become complex.

This is an iterative process. It is derived from considering one step of the standard shifted QR algorithm on the matrix AB^{-1}, which is never formed. On average two iterations per eigenvalue are required. Eigenvectors can be found by the process of *inverse iteration*. Further details can be found in the original paper [37] and [24].

The are two major disadvantages of the QZ algorithm. Firstly it does not take advantage of any symmetry, destroying it at the first step. This means we are not guaranteed real eigenvalues in floating point arithmetic. Secondly the QZ algorithm in unsuitable for nonregular matrix pencils. It can fail to identify the nonregular part of the problem [31].

3.3.5 The MDR Algorithm

In [9] Bunse-Gerstner presents an algorithm for solving the symmetric semidefinite generalized eigenvalue problem, the MDR algorithm.

The algorithm attempts to simultaneously diagonalize A and B and is described as closely related to the QR algorithm for real symmetric matrices. Firstly, the pencil is reduced to the form

$$\begin{bmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ A_{12}^{(1)^T} & A_{22}^{(1)} \end{bmatrix} - \lambda \begin{bmatrix} 0 & 0 \\ 0 & D_{22}^{(2)} \end{bmatrix}, \tag{3.3.1}$$

where $D_{22}^{(1)} \in \mathbb{R}^{r \times r}$ is diagonal and $r = \text{rank}(B)$, by any suitable method. The above pencil is a symmetric permutation of that produced by step one of our algorithm.

A proof is given [9, Thm. 2.1] of the existence of a matrix which can diagonalize (3.3.1) by congruence transformations and requires

$$\text{rank}\left(\begin{bmatrix} A_{11}^{(1)} \\ A_{12}^{T} \end{bmatrix}\right) = \text{rank}(A_{11}^{(1)}) \tag{3.3.2}$$

in (3.3.1), and we note that this is a strong condition.

Furthermore, convergence of the algorithm is only guaranteed if $A_{11}^{(1)}$ is nonsingular. This convergence condition is used in the second step of the MDR algorithm to produce a transformed pencil of the form

$$A^{(2)} = \begin{bmatrix} A_{11}^{(2)} & 0 \\ 0 & A_{22}^{(2)} \end{bmatrix} - \lambda \begin{bmatrix} 0 & 0 \\ 0 & D_{22}^{(2)} \end{bmatrix}, \tag{3.3.3}$$

therefore leaving $A_{22}^{(2)} - \lambda D_{22}^{(2)}$ to be solved and give the finite eigenvalues.

It is recognized in [9] that the MDR algorithm acts like the QR algorithm for the symmetric eigenvalue problem $D_{22}^{(2)^{1/2}} A_{22}^{(2)} D_{22}^{(2)^{1/2}} x = \lambda x$. Thus the MDR algorithm cannot solve nonregular pencils with the convergence condition (3.3.2).

The MDR algorithm then transforms $A_{22}^{(2)}$ to a tridiagonal matrix, T, and goes on to solve

$$(T - \lambda D_{22}^{(2)})x = 0$$

using shifts to accelerate convergence.

3.3.6 The GUPTRI Algorithm

The Algorithm

The GUPTRI (Generalized UPper TRIangular) algorithm [15], [16] aims to capture the structure of the Kronecker Canonical Form (KCF) (1.2.7) of the matrix pencil. Stable computation of the KCF cannot be guaranteed as the transformation matrices may be arbitrarily ill-conditioned. So the GUPTRI algorithm transforms a matrix pencil to a 5-by-5 block upper trapezoidal pencil, each diagonal block is such that its KCF consists only of one type of block.

The GUPTRI algorithm applies unitary transformations $P \in \mathbb{C}^{m \times m}$ and $Q \in \mathbb{C}^{n \times n}$ to the general pencil $A - \lambda B \in \mathbb{C}^{m \times n}$, to generate the *generalized Schur-staircase form*:

$$P^*(A-\lambda B)Q = \begin{bmatrix} A_R-\lambda B_R & X & X & X & X \\ 0 & A_Z-\lambda B_Z & X & X & X \\ 0 & 0 & A_F-\lambda B_F & X & X \\ 0 & 0 & 0 & A_I-\lambda B_I & X \\ 0 & 0 & 0 & 0 & A_L-\lambda B_L \end{bmatrix},$$

where an X denotes a nonzero matrix of conformable dimensions. This is a block upper trapezoidal matrix and the blocks are block upper trapezoidal themselves:

$A_R-\lambda B_R$ is such that its KCF has only L_j blocks.

$A_Z-\lambda B_Z$ is such that its KCF has only $J_j(0)$ blocks.

$A_F-\lambda B_F$ is such that its KCF has only $J_j(\gamma)$, $\gamma \neq 0$, blocks.

$A_I-\lambda B_I$ is such that its KCF has only N_j blocks.

$A_L-\lambda B_L$ is such that its KCF has only L_j^T blocks.

There are two reductions. Firstly the pencil is reduced to *RZ (Right Zero)-staircase form* and a remaining block. That is for unitary matrices $U_1 \in \mathbb{C}^{m\times m}$ and $V_1 \in \mathbb{C}^{n\times n}$

$$U_1^*(A-\lambda B)V_1 = \left.\begin{bmatrix} A_R-\lambda B_R & X & X \\ 0 & A_Z-\lambda B_Z & X \\ 0 & 0 & A_{33}-\lambda B_{33} \end{bmatrix}\right\} \text{RZ-staircase.}$$

This is a achieved in a finite number of transformations. At the kth step the number and dimension of L_{k-1} and $J_k(0)$ blocks is determined, and these values are returned by the algorithm.

The second reduction generates the *LI (Left Infinity)-staircase* part of the pencil. It acts on

$$\mu A_{33} - B_{33} \in \mathbb{C}^{s\times t}, \quad \lambda - 1/\mu.$$

Unitary matrices $U_2 \in \mathbb{C}^{s\times s}$ and $V_2 \in \mathbb{C}^{t\times t}$ are found such that

$$U_2^*(A_{33}-\lambda B_{33})V_2 = \left[\begin{array}{ccc} A_{11}-\lambda B_{11} & X & X \\ 0 & A_I-\lambda B_I & X \\ 0 & 0 & A_L-\lambda B_L \end{array}\right] \Bigg\} \text{ LI-staircase.}$$

The number and dimension of N_j and L_j blocks are computed and returned.

Finally, $A_{11}-\lambda B_{11}$ is transformed to reveal the finite nonzero eigenvalues using the QZ algorithm

$$Q^*(A_{11}-\lambda B_{11})Z = A_F - \lambda B_F.$$

Deciding Rank

The GUPTRI algorithm uses the singular value decomposition to decide numerical rank during the RZ and LI reductions. The algorithm uses two tolerance values, ϵ and gap. The numerical rank of a matrix, r, is defined for singular values σ_i, $\sigma_i > \sigma_{i-1}$, by

$$\sigma_{n-r+1} > \epsilon\sigma_n, \quad \text{and} \quad \sigma_{n-r+1} > gap * \sigma_{n-r}.$$

If these two conditions are satisfied the rank is taken as r, and $\sigma_1, \ldots, \sigma_{n-r+2}$ are set to zero. If the second condition fails σ_{n-r} is also set to zero and the process is repeated. If it is never satisfied then the numerical rank is not defined for that matrix.

3.4 Options for Rank Revealing Decompositions

Our aim is to have RRDs with minimal flop count while maintaining numerical stability and correctly revealing rank. The following tables list possible

RRDs that can be selected in a MATLAB implementation of our algorithm; see `ssgep.m` in Appendix C.

Note since we are not concerned with timings we are using MATLAB's implementation of the RRDs. For built in functions MATLAB calls LAPACK routines where possible. However, for the pivoted Cholesky factorization we call `cholp.m` from [28]. In an implementation outside of MATLAB we would call the codes we developed in Chapter 2.

3.4.1 $B = X_1 D X_1^T$

At step one we perform the transformation

$$\overline{X}_1^{-1} B \overline{X}_1^{-T} = \mathrm{diag}(D_1, 0).$$

Two possible rank revealing decompositions are given in Table 3.4.1. Here

$$L = \begin{bmatrix} L_{11} & 0 \\ L_{12} & 0 \end{bmatrix}, \quad \overline{L} = \begin{bmatrix} L_{11} & 0 \\ L_{12} & I_{r_2} \end{bmatrix}.$$

Name / Eqn. No./ `rrd1` in `ssgep.m`	Factorization of B	flops	$\overline{X}_1^{-1}$
Cholesky Factorization / (2.1.1) / `'chol'`	$B = PLL^TP^T$, $D_1 = I_{r_1}$	$r_1^2 n - \frac{2}{3} r_1^3$	$\overline{L}^{-1} P^T$
Spectral Decomposition / (1.5.3) / `'spec'`	$B = Q \begin{bmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{bmatrix} Q^T$ $D_1 = \Lambda_1$	$4n^3$	Q^T

Table 3.4.1: Options for `rrd1` in `ssgep.m`.

3.4.2 $A_{22}^{(2)} = X_3 D X_3^T$

At step 3 we perform the transformation

$$\overline{X}_3^{-1} A_{22}^{(2)} \overline{X}_3^{-T} = \mathrm{diag}(\Phi_{11}, 0).$$

Two possible rank revealing decompositions are given in Table 3.4.2. Here L and $\overline{L}$ are as defined above.

Name / Eqn. No./ `rrd3` in `ssgep.m`	Factorization of $A_{22}^{(2)}$	flops	$\overline{X}_3^{-1}$
Spectral Decomposition / (1.5.3) / `'spec'`	$A_{22}^{(2)} = Q \begin{bmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{bmatrix} Q^T,$ $D_{22}^{(3)} = \Lambda_1$	$4r_2^3$	Q^T
$LQDQ^TL^T$ Factorization / (1.5.6) / `'ldlt'`	$A_{22}^{(2)} = P^T L Q \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} Q^T L^T P,$ $D_{22}^{(3)} = D_1$	$r_3^2 r_2$ $-\frac{2}{3} r_3^3$	$Q^T \overline{L}^{-1} P$

Table 3.4.2: Options for `rrd3` in `ssgep.m`.

3.4.3 $A_{13}^{(3)} = X_4 D Z_4$

Finally at step four we wish to perform the transformation

$$\overline{X}_4^{-1} A_{13}^{(3)} \overline{Z}_4^{-1} = \mathrm{diag}(A_{14}^{(4)}, 0),$$

where $A_{14}^{(4)}$ can be triangular or diagonal. Four possible rank revealing decompositions are given in Table 3.4.3. Here

$$\overline{R} = \begin{bmatrix} \mathrm{diag}(\mathrm{diag}(R_{11})^{-1}) R_{11} & \mathrm{diag}(\mathrm{diag}(R_{11})^{-1}) R_{12} \\ 0 & I_{r_4 - r_5} \end{bmatrix}.$$

Name / Eqn. No./ `rrd4` in `ssgep.m`	Factorization of $A_{13}^{(3)}$	flops	$\overline{X}_4^{-1}$, $\overline{Z}_4^{-1}$
Unpivoted QR Factorization / (1.4.1) / `'qr'`	$A_{13}^{(3)} = QR$, $G_{14}^{(4)} = R$	$2r_4^2(r_1 - r_4/3)$	$\overline{X}_4^{-1} = Q^T$, $\overline{Z}_4^{-1} = I_{r_4}$
Pivoted QR Factorization / (1.5.1) / `'qrp'`	$A_{13}^{(3)} = Q \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} P^T$, $G_{14}^{(4)} = \mathrm{diag}(R_{11})$	$4(r_1 r_4 r_5 + r_1^2 r_5)$ $-2r_5^2(\frac{3r_1}{2} + r_4)$ $+\frac{5}{3}r_5^3$	$\overline{X}_4^{-1} = Q^T$, $\overline{Z}_4^{-1} = P\overline{R}^{-1}$
Complete Orth. Decomposition / (1.5.2) / `'cod'`	$A_{13}^{(3)} = U \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix} V$, $G_{14}^{(4)} = R_1$	$4(r_1 r_4 r_5 + r_1^2 r_5)$ $-r_5^2(3r_1 + 7r_4)$ $-\frac{5}{3}r_5^3$ $+\frac{4}{3}r_4^3 - 4r_4^2 r_5$	$\overline{X}_4^{-1} = U^T$, $\overline{Z}_4^{-1} = V^T$
Singular Value Decomposition / (1.5.4) / `'svd'`	$A_{13}^{(3)} = U \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} V^T$, $G_{14}^{(4)} = \Sigma_1$	$4r_1^2 r_4 + 8r_1 r_4^2$ $+9r_4^3$	$\overline{X}_4^{-1} = U^T$, $\overline{Z}_4^{-1} = V$

Table 3.4.3: Options for `rrd4` in `ssgep.m`.

3.4.4 Calling `ssgep.m`

The call to `ssgep.m` takes the form `[U,D,r] = ssgep(A,B,rrd1,rrd3,rrd4, reltol,normtol)`. The arguments are as follows:

- `A` is the symmetric matrix A in $Ax - \lambda Bx$.
- `B` is the symmetric matrix B in $Ax - \lambda Bx$.
- `rrd1` specifies the RRD at step one, see Table 3.4.1.
- `rrd3` specifies the RRD at step three, see Table 3.4.2.
- `rrd4` specifies the RRD at step four, see Table 3.4.3.
- `reltol` specifies the tolerance for RRDs. The default is the unit roundoff, u. Then in $A = XDX^T \in \mathbb{R}^{n\times n}$, where the elements in D, d_{ii}, are

decreasing in magnitude, d_{ii}, $i = r+1{:}\,n$, are taken as zero if

$$d_{r+1,r+1} \leq \texttt{reltol} * n * d_{11},$$

and the rank is taken as r. Similarly for $A = XGZ \in \mathbb{R}^{m\times n}$, $G(r+1{:}\,m, r+1{:}\,n)$ are set to zero if

$$g_{r+1,r+1} \leq \texttt{reltol} * \max(m,n) * g_{11}.$$

- `normtol` specifies the tolerance for deciding if $A_{13}^{(3)} \in \mathbb{R}^{r_1\times r_4}$ is to be considered zero, if

$$\|A_{13}^{(3)}\|_F \leq \texttt{normtol} * \max(r_1, r_4) * \|A^{(3)}\|_F$$

then we set $A_{13}^{(3)} = 0$. The default is u.

With the following calls we can implement the Fix and Heiberger, Parlett and Cao algorithms described earlier:

- The Fix & Heiberger algorithm can be implemented with the call:

```
[U,D,r] = ssgep(A,B,'spec','spec','qr',reltol,normtol)
```

- The Parlett algorithm can be implemented with the call:

```
[U,D,r] = ssgep(A,B,'spec','spec','svd',reltol,normtol)
```

- The above is also a call to a Cao algorithm, and so is:

```
[U,D,r] = ssgep(A,B,'spec','spec','cod',reltol,normtol)
```

3.4.5 Operation Count of our Algorithm

In this section we compare the flops of various permutations of RRDs in our algorithm.

If we perform only steps one and two, that is $A_{22}^{(1)}$ has full rank, then performing the transformations and solving the symmetric eigenvalue problem involve

$$4n^3 + 2n^2 r_1 + 4r_1^3 + 2r_1^2 r_2 + 4r_1 r_2^2$$

flops. Then there are further flops to perform the RRDs at step one and three. The RRD with the minimum and maximum flop counts are given in Table 3.4.4.

We can see that if we choose a spectral decomposition in favour of the pivoted Cholesky factorization at step one or the LDL^T at step three we have twelve times as many flops.

There are other factors to consider such as the proportion of computation performed by Level 2 or 3 BLAS routines. To ensure flops are a reasonable indication of speed, we timed the LAPACK routines for the factorizations, as well as `LEV3PCHOL` for the pivoted Cholesky factorization.

The tests were performed in Fortran 77 on a 2545MHz AMD Athlon running a hybrid version of Red Hat Linux 8 and 9 with kernel 2.4.20. The times given are an average of three timings. We generated random symmetric matrices of dimension 1000. The timings are shown in Table 3.4.4.

RRD	Flops	Time (secs)
Spectral Decomposition	$4n^3$	8.18
Cholesky Factorization	$n^3/3$	0.85
LDL^T Factorization	$n^3/3$	0.50

Table 3.4.4: Flops compared with computation time of RRDs.

The timings show that the speedup of our algorithm may not be quite as good as the flops counts predict for the Cholesky factorization, but better for

the LDL^T factorization.

If we perform all steps of our algorithm, but with $A_{13}^{(3)}$ having full rank then the transformations and solving the symmetric eigenvalue problem involve

$$4n^3 + 4r_1^3 + 2r_1^2 r_3 + 2r_2^2 r_1 + 2r_6(n^2 + r_1^2 r_2^2 + r_3 r_6 + r_4^2 + r_5 r_6 + r_5 r_3 + 2r_6^2)$$

flops. Then there are further flops to perform the RRDs at steps one, three and four.

The overall operation count for our algorithm depends on the rank of the matrices we factorize. However, a worst case is approximately $18n^3$ flops. The QZ algorithm requires $46n^3$ flops to compute the eigenvalues and eigenvectors. The GUPTRI algorithm requires $O(n^4)$ flops in the nonregular case for which it is used. Thus our algorithm should perform better than the QZ and GUPTRI algorithm for speed, and the choice of RRD can decrease computation time further. We next look at the stability of our algorithm.

3.5 Numerical Experiments

The tests were performed with MATLAB 6.5 on a 2545MHz AMD Pentium running a hybrid version of Red Hat Linux 8 and 9 with kernel 2.4.20. The unit roundoff $u \approx$ 1.1e-16. The relative normwise backward error for a computed finite eigenvalue, $\widehat{\lambda}$, and its corresponding eigenvector, $\widehat{x}$, of $Ax = \lambda Bx$ is defined to be

$$\gamma(\widehat{x}, \widehat{\lambda}) = \min\{\epsilon : (A + \Delta A)\widehat{x} = \widehat{\lambda}(B + \Delta B)\widehat{x}, \qquad \|\Delta A\|_2 \leq \epsilon \|A\|_2, \quad \|\Delta B\|_2 \leq \epsilon \|B\|_2\}.$$

We evaluate this by the expression [25]

$$\gamma(\widehat{x}, \widehat{\lambda}) = \frac{\|r\|_2}{(|\widehat{\lambda}|\|B\|_2 + \|A\|_2)\|\widehat{x}\|_2}, \tag{3.5.1}$$

where $r = \widehat{\lambda}B\widehat{x} - A\widehat{x}$ is the residual.

For an infinite eigenvalue, $\beta = 0$, this simplifies to

$$\gamma(\widehat{x}, \widehat{\lambda}) = \frac{\|Bx\|_2}{\|B\|_2\|\widehat{x}\|_2}, \tag{3.5.2}$$

We also use the relative residual

$$\rho(\widehat{X}, \widehat{\Lambda}) := \frac{\|A\widehat{X} - B\widehat{X}\widehat{\Lambda}\|_2}{\|A\|_2\|\widehat{X}\|_2 + \|B\|_2\|\widehat{X}\|_2\|\widehat{\Lambda}\|_2}, \tag{3.5.3}$$

where $\widehat{X}$ is a matrix of eigenvectors corresponding to finite eigenvalues and $\widehat{\Lambda}$ is a diagonal matrix of finite eigenvalues.

We present the comparison of these errors, for finite eigenvalues, by performance profiles. Where the computed error was less than the unit roundoff we set it to `eps/2`, the unit roundoff in MATLAB.

We test two implementations of our algorithm, those that require the maximum and minimum flops as described above. The first implementation uses only orthogonal RRDs so all transformations are orthogonal with the exception of U_2 at step two, and we saw the advantage of orthogonal transformations in Section 1.3.2. The second uses RRDs that are not orthogonal, using the pivoted Cholesky factorization, the stability of which is discussed in Section 2.6 and the LDL^T factorization discussed in [29].

3.5.1 Industrial Example

This is a fluid flow problem from a dynamical analysis in structural engineering and was taken from the Harwell-Boeing Collection available at Matrix Market [34]. The matrices are from the 'BCSSTRUC1' data set and the mass matrix, M, is 'BCSSTM13' and the stiff matrix, K, is 'BCSSTK13'. The dimension of the matrices is 2003 and K is positive definite so step four of our algorithm will not be performed.

Using default tolerances, the call

```
[U,D,r] = ssgep(K,M,'spec','spec','svd')
```

gave a maximum $\gamma(\widehat{x}, \widehat{\lambda})$ of 8.246e-16 and for

```
[U,D,r] = ssgep(K,M,'chol','ldlt','svd')
```

a maximum $\gamma(\widehat{x}, \widehat{\lambda})$ of 7.607e-16. The matrices here are very well conditioned.

This was the largest data set available to the author. Also, since K is positive definite, we generate our own test matrices to fully explore the behavior of our algorithm and other methods.

3.5.2 Regular Pencil Examples

Here we generate some random regular pencils and compare implementations of our algorithm with the QZ algorithm.

We construct a set of test matrices as follows:

B is set to be

$$Q^T \text{diag}(\lambda_1,\ \lambda_2\ , \ldots,\ \lambda_{r_1},\ 0, \ldots, 0) Q,$$

where Q is a random orthogonal matrix.

We construct A such that

$$A = \begin{array}{c} \\ r_1 \\ r_2 \end{array} \begin{array}{c} \begin{array}{cc} r_1 & r_2 \end{array} \\ \begin{bmatrix} A_{11}^{(2)} & A_{12}^{(2)} \\ A_{12}^{(2)^T} & A_{22}^{(2)} \end{bmatrix} \end{array},$$

where

- $A_{11}^{(2)}$ is a random symmetric matrix.

- $A_{22}^{(2)}$ has full rank, we compute

$$A_{22}^{(2)} = Q \, \mathrm{diag}(\bar{\lambda}_1, \ \bar{\lambda}_2, \ \ldots, \ \bar{\lambda}_{r_2}) \, Q^T,$$

  where $\bar{\lambda} \neq 0$ and $Q \in \mathbb{R}^{r_2 \times r_2}$ is a random orthogonal matrix.

- $A_{12}^{(2)}$ is a random matrix.

We chose the nonzero eigenvalues, λ_i (and $\bar{\lambda}_i$) , to be geometrically decreasing

$$\lambda_i = \alpha^{i-1}, \quad 1 \leq i \leq r, \quad \alpha \leq 1,$$

and α was chosen to vary $\kappa_2 = \lambda_1/\lambda_r$. See `gen_data.m` in Appendix C.

For each case we constructed a set of 48 matrices, with $n = 500$, by using every combination of:

$$\begin{aligned} \kappa_2(B) &= \{1e{+}3, \ 1e{+}6, \ 1e{+}9, \ 1e{+}12\}, \\ \kappa_2(A_{22}) &= \{1e{+}3, \ 1e{+}6, \ 1e{+}9, \ 1e{+}12\}, \\ \text{rank of } B \text{ and } A_{22} \text{ (as \%)} &= \{30, \ 50, \ 70\}. \end{aligned}$$

We tested three algorithms:

- `[U,D,r] = ssgep(A,B,'chol','ldlt','qrp')`
- `[U,D,r] = ssgep(A,B,'spec','spec','svd')`
- `[U,D] = eig(A,B)`, (the QZ algorithm)

and computed the maximum $\gamma(\widehat{x}, \widehat{\lambda})$ and $\rho(\widehat{X}, \widehat{\Lambda})$ for finite eigenvalues and the condition number of the overall transformation matrix. We represent the results by performance profiles.

Figure 3.5.1 shows the performance profile for the maximum $\gamma(\widehat{x}, \widehat{\lambda})$ of each algorithm on each test problem. For approximately 80% of the test

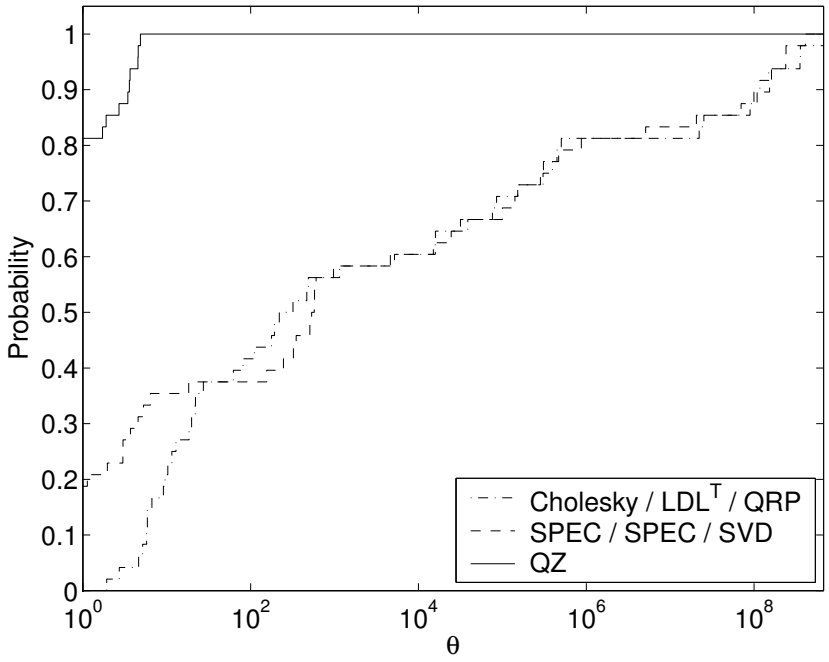

Figure 3.5.1: Performance profile for maximum backward error, $\gamma(\widehat{x}, \widehat{\lambda})$, of computed finite eigenvalues and eigenvectors.

problems the QZ algorithm performed best, where the smallest error was 4.27e-16. Around 20% of the problems gave the smallest value of the maximum $\gamma(\widehat{x}, \widehat{\lambda})$ with the implementation of our algorithm with orthogonal RRDs, shown (SPEC / SPEC / SVD). In comparison the implementation with nonorthogonal RRDs, shown (Cholesky / LDL^T / QRP), performed poorly, never achieving the smallest error for a particular test problem. However, the profile is very close to that with the orthogonal RRDs for $\theta > 100$.

The values of $\rho(\widehat{X}, \widehat{\Lambda})$ were more favourable for implementations of our algorithm, and are shown in Figure 3.5.2. The proportion of values that were smallest for each problem are about the same as for $\gamma(\widehat{x}, \widehat{\lambda})$. However, every result was within a factor of 25 of the smallest.

We also measured the condition number of the overall transformation, U,

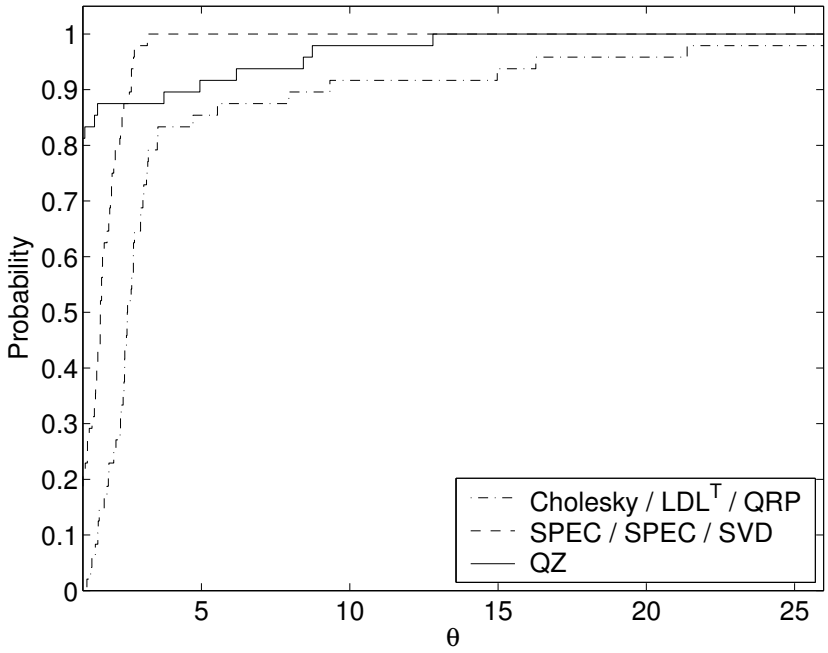

Figure 3.5.2: Performance profile for the relative residual, $\rho(\widehat{X}, \widehat{\Lambda})$, of computed finite eigenvalues and eigenvectors.

such that

$$U^T(A - \lambda B)U = A^{(4)} - \lambda B^{(4)},$$

which for our algorithm is

$$U = U_4 U_3 U_2 \overline{X}_1^{-T},$$

and

$$\begin{aligned} \kappa_2(U) &= \|U\|_2 \|U^{-1}\|_2 \\ &\leq \|U_4\|_2 \|U_4^{-1}\|_2 \|U_3\|_2 \|U_3^{-1}\|_2 \|U_2\|_2 \|U_2^{-1}\|_2 \|\overline{X}_1^T\|_2 \|\overline{X}_1^{-T}\|_2. \end{aligned}$$

Unsurprisingly the condition of Q and Z for the QZ algorithm was 1 or very close to 1. For our algorithm with the orthogonal RRDs $\kappa_2(U) \leq \|U_2\|_2 \|U_2^{-1}\|_2 = \sqrt{\kappa_2(B)}$ as the only nonorthogonal transformation matrix

$U_2 = \text{diag}(D_1^{-1/2}, I_{r_2})$, where D_1 holds the square root of the eigenvalues of B. This gives the steps in Figure 3.5.3. Our algorithm with nonorthogonal RRDs gave a value as large as 1e+8 and also steps up as the condition number of B is increased.

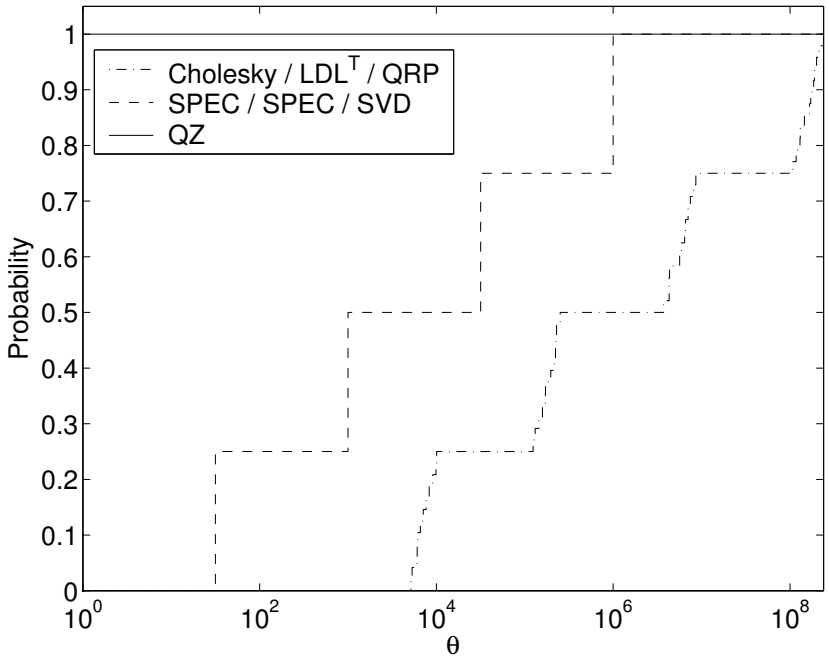

Figure 3.5.3: Performance profile for condition number, $\kappa_2(U)$, of overall transformation matrix U.

The QZ algorithm destroys symmetry so real eigenvalues are not guaranteed. In the above examples the QZ algorithm returned complex eigenvalues in 7 out of the 48 test matrices. The imaginary part was of order 1e-14 or 1e-15.

3.5.3 Nonregular Pencil Examples

Here we generate some random nonregular pencils and compare implementations of our algorithm with a Fortran 77 implementation of the GUPTRI

algorithm called via a MEX interface.

We construct a set of test matrices as follows:

$$B = \begin{bmatrix} I_{r_1} & 0 \\ 0 & 0 \end{bmatrix}$$

With this B we can specify A so as to predict (in exact arithmetic) r_i, $i = 1{:}\,7$, as the transformation at step 1 will not alter the spectrum of the submatrices of A set out below.

We construct A such that

$$A = \begin{array}{c} \\ r_1 \\ r_2 \end{array} \begin{array}{c} \begin{array}{cc} r_1 & r_2 \end{array} \\ \begin{bmatrix} A_{11}^{(2)} & A_{12}^{(2)} \\ A_{12}^{(2)^T} & A_{22}^{(2)} \end{bmatrix} \end{array},$$

where

- $A_{11}^{(2)}$ is a random symmetric matrix.
- $A_{22}^{(2)}$ has r_3 nonzero eigenvalues and r_4 zero eigenvalues, we compute

  $$Q\ \mathrm{diag}(\lambda_1,\ \lambda_2,\ \ldots,\ \lambda_{r_3},\ 0, \ldots, 0)\ Q^T,$$

  where $\lambda \neq 0$ and $Q \in \mathbb{R}^{r_2 \times r_2}$ is a random orthogonal matrix.
- $A_{12}^{(2)}$ has r_5 nonzero singular values and $\min(r_1, r_2) - r_5$ zero singular values, we compute

  $$Q_1\ \mathrm{diag}(\sigma_1,\ \sigma_2,\ \ldots,\ \sigma_{r_5},\ 0, \ldots, 0)\ Q_2,$$

  where $\sigma \neq 0$ and $Q_1 \in \mathbb{R}^{r_1 \times r_1}$, $Q_2 \in \mathbb{R}^{r_2 \times r_2}$ are random orthogonal matrices.

We chose the nonzero eigenvalues, λ_i (and singular values, σ_i) again geometrically decreasing

$$\lambda_i = \alpha^{i-1}, \quad 1 \leq i \leq r, \quad \alpha \leq 1,$$

where α was chosen to vary $\kappa_2 = \lambda_1/\lambda_r$. Again $n = 500$.

First we set $\kappa_2 = 1$ and

$$\text{rank of } B,\ A_{22} \text{ and } A_{12} \text{ (as \%)} = \{30,\ 50,\ 70\}$$

and repeated this 16 times to give a set of 48 test matrices.

We tested three algorithms

- `[U,D,r] = ssgep(A,B,'chol','ldlt','qrp')`
- `[U,D,r] = ssgep(A,B,'spec','spec','svd')`
- The GUPTRI algorithm with default tolerances.

and again computed the maximum $\gamma(\widehat{x}, \widehat{\lambda})$ and $\rho(\widehat{X}, \widehat{\Lambda})$ for finite eigenvalues and the condition number of the overall transformation matrix.

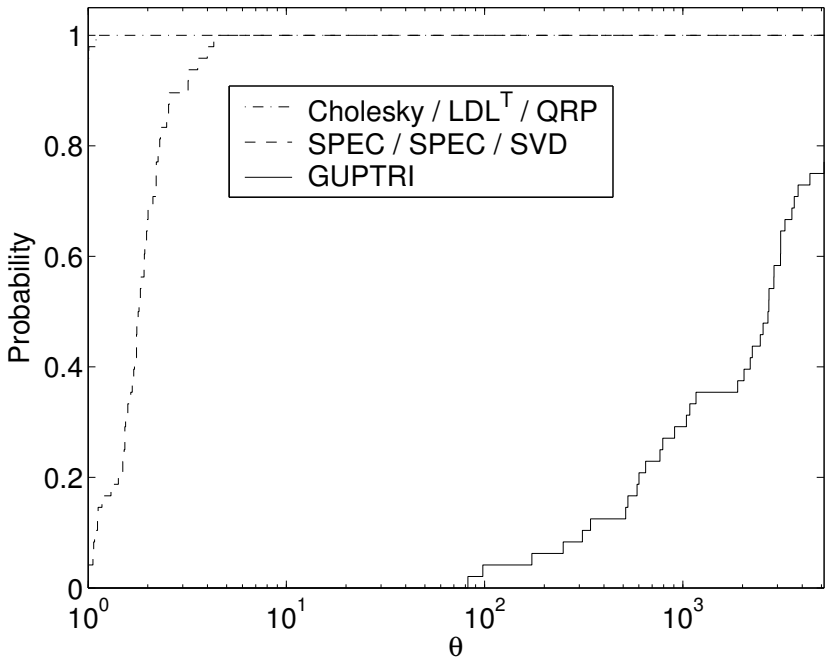

Figure 3.5.4: Performance profile for maximum backward error, $\gamma(\widehat{x}, \widehat{\lambda})$, of computed finite eigenvalues and eigenvectors.

The GUPTRI algorithm 'failed' on 10 of the 48 test problems. It failed in the sense that it gave one or more zero eigenvalues not predicted by the test problem and also gave L_j blocks with $j > 0$ in the KCF, contradicting the theory of Cao [11]. This failure then affected the computation of the remaining nonzero finite eigenvalues. On inspection some eigenvalues were close to those computed by our algorithm, but some were not. The maximum value of $\gamma(\widehat{x}, \widehat{\lambda})$ in these cases was of order 1e-1. These results were omitted from the following performance profiles.

From Figure 3.5.4 we can see the GUPTRI algorithm never achieved the smallest maximum value for $\gamma(\widehat{x}, \widehat{\lambda})$. Our algorithm with nonorthogonal RRDs gave the samllest value for 100% of the problems. However, the implementation of our algorithm with orthogonal RRDs was within a factor of 5 of the figures for nonorthogonal RRDs.

The performance profile for the values of $\rho(\widehat{X}, \widehat{\Lambda})$ is shown in Figure 3.5.5. The results here are very similar to those of $\gamma(\widehat{x}, \widehat{\lambda})$.

The condition number of the overall transformation matrix is shown in Figure 3.5.6. For our algorithm with orthogonal RRDs we should have $\kappa_2(U) \leq \|U_2\|_2\|U_2^{-1}\|_2 = \sqrt{\kappa_2(B)} = 1$, and indeed the value was 1 or very close to 1. Also the condition numbers of the transformation matrices returned by GUPTRI were 1 or very close to 1. For the implementation of our algorithm with nonorthogonal RRDs we had condition numbers as high as 1e+3.

We repeated the experiments with $\kappa_2 = 10$ and the GUPTRI algorithm now failed 40 times out of 48. Failure of the GUPTRI algorithm to identify the KCF is discussed in [10].

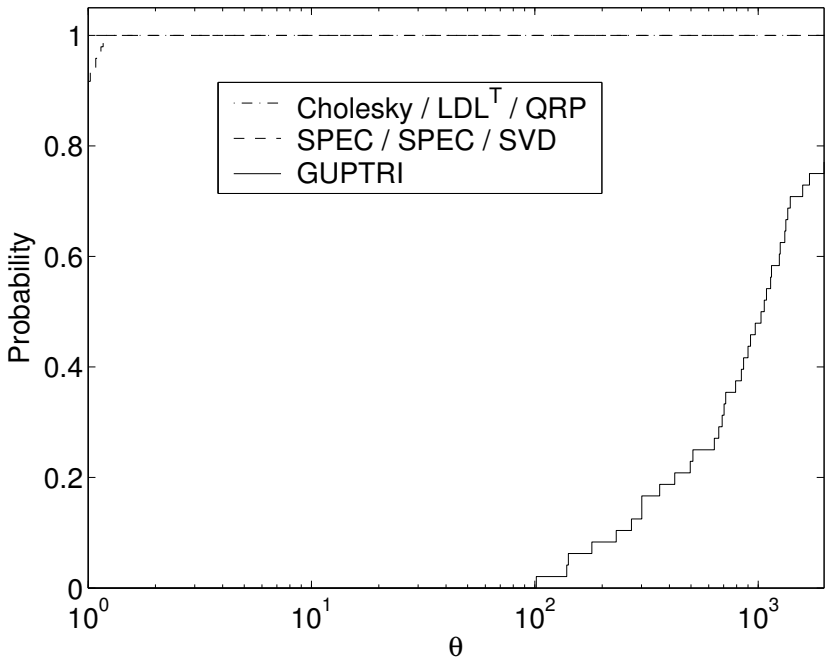

Figure 3.5.5: Performance profile for the relative residual, $\rho(\widehat{X}, \widehat{\Lambda})$, of computed finite eigenvalues and eigenvectors.

3.6 Conclusions

There is a trade-off with implementations of our algorithm. The implementation with nonorthogonal RRDs can require far fewer flops than using orthogonal RRDs. However, the backward error is generally larger than with orthogonal transformation matrices, although the difference is not always significant. With the nonregular test problems the nonorthogonal RRDs implementation fared slightly better, although this was with submatrices with a condition number of 1. We have only tested two implementations of our algorithm and there are many other combinations of RRD possible.

The QZ algorithm gives the best backward error result, compared to both implementations of our algorithm, more often for regular pencils but requires at least three times as many flops. The QZ algorithm also destroys symmetry

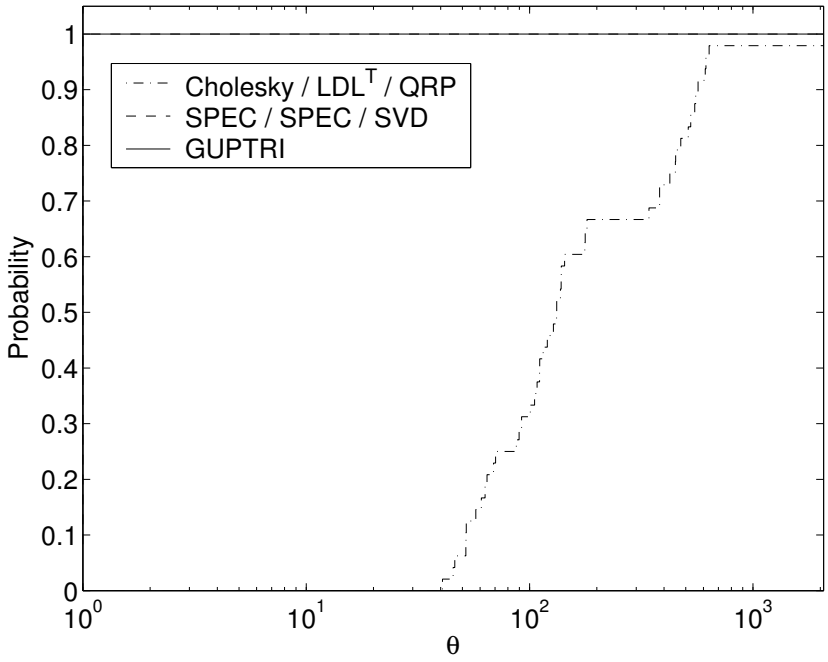

Figure 3.5.6: Performance profile for condition number, $\kappa_2(U)$, of overall transformation matrix U. Note the last two lines are coincident.

and can produce eigenvalues with a small imaginary part and it cannot be relied upon for nonregular pencils.

The GUPTRI algorithm can fail for test problems with a small condition number and requires flops of $O(n^4)$ for nonregular pencils. The algorithm always produced a larger error than implementations of our algorithm, and was at least a factor of 100 greater. Thus any implementation of our algorithm is more efficient and more stable than the GUPTRI algorithm for out test problems. Also, GUPTRI destroys the symmetry of the problem.

In both our algorithm and the GUPTRI algorithm we are making rank decisions. It is unclear what the effect of these are on the overall performance of the algorithms, and this is an open question.

There are other issues we have not addressed. There may be cancellation

errors in forming the symmetric eigenvalue problem

$$A_{22}^{(4)} - A_{23}^{(4)} D_{33}^{(4)^{-1}} A_{23}^{(4)^T},$$

and $D_{33}^{(4)^{-1}}$ may be ill conditioned depending on our choice of tolerance for the RRD.

Further work is required to address and understand these problems.

3.7 The Symmetric Indefinite Generalized Eigenvalue Problem

We briefly describe here how our algorithm can be adapted for the generalized eigenvalue problem

$$Ax = \lambda Bx \tag{3.7.1}$$

in the case where A and B are real and symmetric and B is now indefinite and may be singular. We consider the algorithm as a deflation strategy.

3.7.1 Step One

First we diagonalize the matrix B by computing an RRD

$$B = X_1 \mathrm{diag}(D_1, D_2) X_1^T,$$

where $D_1 \in \mathbb{R}^{r_1 \times r_1}$ and $D_2 \in \mathbb{R}^{r_2 \times r_2}$, with r_1 chosen minimally so that $\|D_2\|_2 \leq \eta \|D_1\|_2$. We set $D_2 = 0$. We thus transform the pencil $A - \lambda B$

$$A, B \xrightarrow{\overline{X_1}^T} A^{(1)}, B^{(1)} = \begin{array}{c} \\ r_1 \\ r_2 \end{array} \begin{array}{c} \begin{array}{cc} r_1 & r_2 \end{array} \\ \begin{bmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ A_{12}^{(1)^T} & A_{22}^{(1)} \end{bmatrix} \end{array}, \mathrm{diag}(D_1, 0).$$

Since all the diagonal elements of D_1 are not guaranteed to be all of the same sign, we cannot reduce it to the identity matrix as in the semidefinite case.

If B has full rank then $r_2 = 0$ and we can now solve the regular symmetric indefinite eigenvalue problem

$$A^{(2)}z = \lambda D_1 z.$$

or the nonsymmetric standard eigenvalue problem

$$D_1^{-1}A^{(2)}z = \lambda z,$$

by any suitable method. If B is rank deficient we proceed to step two.

3.7.2 Step Two

We now reduce $A_{22}^{(1)}$ to diagonal form. Compute the RRD

$$A_{22}^{(1)} = X_2 \Phi_2 X_2^T,$$

where the diagonal matrix

$$\Phi_2 = \begin{array}{c} \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{cc} r_3 & r_4 \end{array} \\ \begin{bmatrix} \Phi_{11} & 0 \\ 0 & \Phi_{22} \end{bmatrix} \end{array},$$

where $r_2 = r_3 + r_4$ and r_3 is chosen minimally so that $\|\Phi_{22}\|_2 \leq \eta \|\Phi_{11}\|_2$. We set $\Phi_{22} = 0$ and define $U_2 = \mathrm{diag}(I_{r_1}, \overline{X}_3^{-T})$ and hence transform

$$A^{(1)}, B^{(1)} \xrightarrow{U_2} A^{(2)}, B^{(2)},$$

where

$$A^{(2)} = \begin{array}{c} \\ r_1 \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{ccc} r_1 & r_3 & r_4 \end{array} \\ \begin{bmatrix} A_{11}^{(2)} & A_{12}^{(2)} & A_{13}^{(2)} \\ A_{12}^{(2)T} & D_{22}^{(2)} & 0 \\ A_{13}^{(2)T} & 0 & 0 \end{bmatrix} \end{array}, \quad B^{(2)} = B^{(1)} = \mathrm{diag}(D_1, 0),$$

and $D_{22}^{(3)} = \Phi_{11}$.

Now if $r_4 = 0$, the last row and column of $A^{(2)}$ is not present and we have the eigenproblem,

$$\begin{array}{c} \\ r_1 \\ r_2 \end{array} \beta \begin{array}{c} \begin{array}{cc} r_1 & r_2 \end{array} \\ \begin{bmatrix} A_{11}^{(2)} & A_{12}^{(2)} \\ A_{12}^{(2)^T} & D_{22}^{(2)} \end{bmatrix} \end{array} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \alpha \begin{bmatrix} D_1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}. \tag{3.7.2}$$

Solving for w_2 gives $w_2 = -D_{22}^{(2)^{-1}} A_{12}^{(2)^T} w_1$ and hence we obtain the r_1-by-r_1 eigenvalue problem

$$\beta D_1^{-1}(A_{11}^{(2)} - A_{12}^{(2)} D_{22}^{(2)^{-1}} A_{12}^{(2)^T}) w_1 = \alpha w_1. \tag{3.7.3}$$

There are r_1 finite eigenvalues, from (3.7.3), and another r_2 infinite eigenvalues corresponding to $\beta = 0$.

If $r_4 > 0$ and $A_{13}^{(3)} = 0$ then the pencil $A - \lambda B$ is nonregular. On removing the last r_4 rows and columns we deflate the nonregular part of the pencil and we again obtain a system of the form (3.7.2), which provides r_1 finite and r_3 infinite eigenvalues.

If neither $r_4 = 0$ nor $A_{13}^{(3)} = 0$ then we proceed to step three.

3.7.3 Step Three

Now we make $A_{13}^{(2)} \in \mathbb{R}^{r_1 \times r_4}$ triangular or diagonal. We compute the RRD

$$A_{13}^{(2)} = X_3 \begin{array}{c} \\ r_5 \\ r_1 - r_5 \end{array} \begin{array}{c} \begin{array}{cc} r_5 & r_4 - r_5 \end{array} \\ \begin{bmatrix} G_{11} & 0 \\ 0 & 0 \end{bmatrix} \end{array} Z_3,$$

where G_{11} is diagonal or triangular. This decomposition will involve some η-negligibility decisions during its computation.

We define $U_3 = \mathrm{diag}(X_3^{-T}, I_{r_3}, \overline{Z}_3^{-1})$, and perform the transformation

$$A^{(2)}, B^{(2)} \xrightarrow{U_3} A^{(3)}, B^{(3)},$$

where $A^{(3)}, B^{(3)}$ is

$$\left[\begin{array}{cc|ccc} A_{11}^{(4)} & A_{12}^{(4)} & A_{13}^{(4)} & G_{14}^{(4)} & 0 \\ A_{12}^{(4)^T} & A_{22}^{(4)} & A_{23}^{(4)} & 0 & 0 \\ \hline A_{13}^{(4)^T} & A_{23}^{(4)^T} & D_{33}^{(4)} & 0 & 0 \\ G_{14}^{(4)^T} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right], \quad \left[\begin{array}{cc|ccc} B_{11}^{(3)} & B_{12}^{(3)} & 0 & 0 & 0 \\ B_{12}^{(3)^T} & B_{22}^{(3)} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right], \quad \begin{array}{l} r_5 \\ r_6 := r_1 - r_5 \\ r_3 \\ r_5 \\ r_7 := r_4 - r_5 \end{array}$$

and $G_{14}^{(4)} = G_{11}$, $D_{33}^{(4)} = D_{22}^{(3)}$. The diagonal matrix D_1 has been made full by the transformation.

After removing the last block row and column of zeros, if necessary, we are left with the following regular eigenproblem,

$$\beta \begin{bmatrix} A_{11}^{(3)} & A_{12}^{(3)} & A_{13}^{(3)} & G_{14}^{(3)} \\ A_{12}^{(3)^T} & A_{22}^{(3)} & A_{23}^{(3)} & 0 \\ A_{13}^{(3)^T} & A_{23}^{(3)^T} & D_{33}^{(3)} & 0 \\ G_{14}^{(3)^T} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \alpha \begin{bmatrix} B_{11}^{(3)} & B_{12}^{(3)} & 0 & 0 \\ B_{12}^{(3)^T} & B_{22}^{(3)} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}. \tag{3.7.4}$$

Solving from the bottom up gives

$$\begin{aligned} \beta v_1 &= 0, \\ \beta v_3 &= -\beta D_{33}^{(3)^{-1}} A_{23}^{(3)^T} v_2, \\ \beta (A_{22}^{(3)} - A_{23}^{(3)} D_{33}^{(3)^{-1}} A_{23}^{(3)^T}) v_2 &= \alpha B_{22}^{(3)} v_2, \\ \beta v_4 &= \alpha B_{12}^{(3)} v_2 - \beta G_{14}^{(4)^{-1}} (A_{12}^{(4)} v_2 + A_{13}^{(4)} v_3). \end{aligned} \tag{3.7.5}$$

We thus have another symmetric indefinite generalized eigenvalue problem, of dimension r_6, and not a standard eigenproblem as in the semidefinite case. We can apply a recursion by restarting the algorithm with (3.7.5). This can continue until the algorithm stops naturally solving a regular problem.

Chapter 4

Updating the QR Factorization and the Least Squares Problem

4.1 Introduction

The linear system,

$$Ax = b,$$

where $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^n$ is *overdetermined* if $m \geq n$. We can solve the *least squares problem*

$$\min_x \|Ax - b\|_2,$$

with A having full rank. We then have with the QR factorization $A = QR$ and with $d = Q^T b$,

$$\begin{aligned} \|Ax - b\|_2^2 &= \|Q^T A - Q^T b\|_2^2 \\ &= \|Rx - d\|_2^2 \\ &= \left\| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x - \begin{bmatrix} f \\ g \end{bmatrix} \right\|_2^2 \\ &= \|R_1 x - f\|_2^2 + \|g\|_2^2, . \end{aligned}$$

where $R_1 \in \mathbb{R}^{n\times n}$ is upper triangular and $f \in \mathbb{R}^n$. The minimum 2-norm solution is then found by solving $R_1 x = f$. The quantity $\|g\|_2$ is the *residual* and is zero in the case $m = n$.

Each row of the matrix A can be said to hold *observations* of the *variables* x_i, $i = 1\colon n$. An example of this is the data fitting problem. Consider the function

$$g(t_i) \approx b_i, \quad i = 1\colon m.$$

The value b_i has been observed at time t_i. We wish to find the function g that approximates the value b_i. In least squares fitting we restrict ourselves to functions of the form

$$g(t) = x_1 g_1(t) + x_2 g_2(t) + \cdots + x_n g_n(t),$$

where the functions $g_i(t)$ we call basis functions, and the coefficients x_i are to be determined. We find the coefficients by solving the least squares problem with

$$A = \begin{bmatrix} g_1(t_1) & g_2(t_1) & \cdots & g_n(t_1) \\ g_1(t_2) & g_2(t_2) & \cdots & g_n(t_2) \\ \vdots & \vdots & & \vdots \\ g_1(t_m) & g_2(t_m) & \cdots & g_n(t_m) \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

Now, it may be required to *update* the least squares solution in the case where one or more observations (rows of A) are added or deleted. For instance we could have a *sliding window* where for each new observation recorded the oldest one is deleted. The observations for a particular time period may be found to be faulty, thus a block of rows of A would need to be deleted. Also, variables (columns of A) may be added or omitted to compare the different solutions. Updating after rows and columns have been deleted is also known as *downdating*.

To solve these updated least squares problems efficiently we have the problem of updating the QR factorization efficiently, that is we wish to find $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the updated A, without recomputing the factorization from scratch. We assume that $\widetilde{A}$ has full rank. We also need to compute $\tilde{d}$ such that

$$\|\widetilde{A}x - \tilde{b}\| = \|\widetilde{R}x - \tilde{d}\|,$$

where $\tilde{b}$ is the updated b corresponding to $\widetilde{A}$ and $\tilde{d} = \widetilde{Q}^T\tilde{b}$.

4.2 Updating Algorithms

In this section we will examine all the cases where observations and variables are added to or deleted from the least squares problem. We derive algorithms for updating the solution of the least squares problem by updating the QR factorization of $\widetilde{A}$, in the case $m \geq n$. For completeness we have also included discussion and algorithms for updating the QR factorization only when $m < n$. In all cases we give algorithms for computing $\widetilde{Q}$ should it be required. We will assume that A and $\widetilde{A}$ have full rank.

Where possible we derive blocked algorithms to exploit the Level 3 BLAS and existing Level 3 LAPACK routines. We include LAPACK style Fortran 77 code for updating the QR factorization in the cases of adding and deleting blocks of columns.

For clarity the sines and cosines for Givens matrices and the Householder vectors are stored in separate vectors and matrices, but could be stored in the elements they eliminate. Wherever possible new data overwrites original data. All unnecessary computations have been avoided, unless otherwise stated.

We give floating point operation counts for our algorithms and compare them to the counts for the Householder QR factorization of $\widetilde{A}$.

Some of the material is based on material in [4] and [24].

4.2.1 Deleting Rows

Deleting One Row

If we wish to update the least squares problem in the case of *deleting* an *observation* we have the problem of updating the QR factorization of A having deleted the kth row, a_k^T. We can write

$$\widetilde{A} = \begin{bmatrix} A(1\colon k-1, 1\colon n) \\ A(k+1\colon m, 1\colon n) \end{bmatrix}$$

and we interpret $A(1\colon 0, 1\colon n)$ and $A(m+1\colon m, 1\colon n)$ as empty rows. We define a permutation matrix P such that

$$PA = \begin{bmatrix} a_k^T \\ A(1\colon k-1, 1\colon n) \\ A(k+1\colon m, 1\colon n) \end{bmatrix} = \begin{bmatrix} a_k^T \\ \widetilde{A} \end{bmatrix} = PQR,$$

and if q^T is the first row of PQ then we can zero $q(2\colon m)$ with $m-1$ Givens matrices, $G(i,j) \in \mathbb{R}^{m\times m}$, so that

$$G(1,2)^T \ldots G(m-1,m)^T q = \alpha e_1, \quad |\alpha| = 1, \tag{4.2.1}$$

since the Givens matrices are orthogonal. And we also have

$$G(1,2)^T \ldots G(m-1,m)^T R = \begin{bmatrix} v^T \\ \widetilde{R} \end{bmatrix},$$

which is upper Hessenberg, so $\widetilde{R}$ is upper trapezoidal.

So we have finally

$$\begin{aligned} PA = \begin{bmatrix} a^T \\ \widetilde{A} \end{bmatrix} &= (PQG(m-1,m)\ldots G(1,2))(G(1,2)^T \ldots G(m-1,m)^T R) \\ &= \begin{bmatrix} \alpha & 0 \\ 0 & \widetilde{Q} \end{bmatrix} \begin{bmatrix} v^T \\ \widetilde{R} \end{bmatrix}, \end{aligned}$$

and

$$\widetilde{A} = \widetilde{Q}\widetilde{R}.$$

Note that the zero column below α is forced by orthogonality. Also note the choice of a sequence of Givens matrices over one Householder matrix. If we were to use a Householder matrix then the transformed R would be full, as H is full, and not upper Hessenberg. We update b by computing

$$G(1,2)^T \ldots G(m-1,m)^T Q^T P b = \begin{bmatrix} \nu \\ \\ \tilde{d} \end{bmatrix}.$$

This gives the following algorithm.

Algorithm 4.2.1 *Given $A = QR \in \mathbb{R}^{m\times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T \widetilde{A} = \widetilde{R} \in \mathbb{R}^{(m-1)\times n}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with the kth row deleted, $1 \leq k \leq m$, and $\tilde{d}$ such that $\|\widetilde{A}x - \tilde{b}\|_2 = \|\widetilde{R}x - \tilde{d}\|_2$, where $\tilde{b}$ is b with the kth element deleted. The residual, $\|\tilde{d}(n+1\colon m-1)\|_2$, is also computed.*

$q^T = Q(k, 1\colon m)$
if $k \neq 1$
    % Permute b
    $b(2\colon k) = b(1\colon k-1)$
end
$d = Q^T b$
for $j = m-1\colon -1\colon 1$
    $[c(j), s(j)] = \mathbf{givens}(q(j), q(j+1))$
    % Update q
    $q(j) = c(j)q(j) - s(j)q(j+1)$
    % Update R if there is a nonzero row
    if $j \leq n$

$$R(j\colon j+1, j\colon n) = \begin{bmatrix} c(j) & s(j) \\ -s(j) & c(j) \end{bmatrix}^T R(j\colon j+1, j\colon n)$$

    end
    % Update d

$$d(j\colon j+1) = \begin{bmatrix} c(j) & s(j) \\ -s(j) & c(j) \end{bmatrix}^T d(j\colon j+1)$$

end
$\widetilde{R} = R(2\colon m, 1\colon n)$
$\tilde{d} = d(2\colon m)$
% Compute the residual
$resid = \|\tilde{d}(n+1\colon m-1)\|_2$

Computing $\widetilde{R}$ requires $3n^2$ flops, versus $2n^2(m-n/3)$ for the Householder QR factorization of $\widetilde{A}$. If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.2 *Given vectors c and s from Algorithm 4.2.1 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{(m-1)\times(m-1)}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with the kth row deleted.*

if $k \neq 1$
  % Permute Q
  $Q(2\colon k, 1\colon m) = Q(1\colon k-1, 1\colon m)$
end
for $j = m-1\colon -1\colon 2$
  $Q(2\colon m, j\colon j+1) = Q(2\colon m, j\colon j+1)\begin{bmatrix} c(j) & s(j) \\ -s(j) & c(j) \end{bmatrix}$
end
% Do not need to update 1st column of Q
$Q(2\colon m, 2) = s(1)Q(2\colon m, 1) + c(1)Q(2\colon m, 2)$
$\widetilde{Q} = Q(2\colon m, 2\colon m)$

Deleting a Block of Rows

If a block of p observations is to be deleted from our least squares problem, equivalent to deleting the p rows $A(k\colon k+p-1, 1\colon n)$ from A, we would like to find an analogous method to Algorithm 4.2.1 that uses Householder matrices, such that if H is a product of p Householder matrices then

$$PA = \begin{bmatrix} A(k\colon k+p-1, 1\colon n) \\ \widetilde{A} \end{bmatrix} = (PQH)(HR) = \begin{bmatrix} I & 0 \\ 0 & \widetilde{Q} \end{bmatrix}\begin{bmatrix} V \\ \widetilde{R} \end{bmatrix}.$$

However as noted in the single row case, HR is full and H is chosen to introduce zeros in Q not R. Thus in order to compute $\widetilde{A} = \widetilde{Q}\widetilde{R}$ and $\tilde{d}$, we need the equivalent of p steps of Algorithm 4.2.1 and Algorithm 4.2.2, since Givens matrices only affect two rows of the matrix they are multiplying, and so we have the following algorithm.

Algorithm 4.2.3 *Given $A = QR \in \mathbb{R}^{m\times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{(m-p)\times n}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with the kth to $(k+p-1)$st rows deleted, $1 \leq k \leq m-p+1$, $1 \leq p < m$, and $\tilde{d}$ such that $\|\widetilde{A}x - \tilde{b}\|_2 = \|\widetilde{R}x - \tilde{d}\|_2$, where $\tilde{b}$ is b with the kth to $(k+p-1)$st elements deleted. The residual, $\|\tilde{d}(n+1\colon m-p)\|_2$, is also computed.*

$W = Q(k\colon k+p-1, 1\colon m)$
if $k \neq 1$
  % Permute b
  $b(p+1\colon k+p-1) = b(1\colon k-1)$
end
$d = Q^T b$
for $i = 1\colon p$
  for $j = m-1\colon -1\colon i$
    $[C(i,j), S(i,j)] = \mathbf{givens}(W(i,j), W(i,j+1))$
    % Update W
    $W(i,j) = W(i,j)C(i,j) - W(i,j+1)S(i,j)$
    $W(i+1\colon p, j\colon j+1) = W(i+1\colon p, j\colon j+1)\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}$
    % Update R if there is a nonzero row
    if $j \leq n+i-1$
      $R(j\colon j+1, j-i+1\colon n) =$
        $\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T R(j\colon j+1, j-i+1\colon n)$
    end
    % Update d
    $d(j\colon j+1) = \begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T d(j\colon j+1)$
  end
end
$\widetilde{R} = R(p+1\colon m, 1\colon n)$
$\tilde{d} = d(p+1\colon m)$
% Compute the residual
$resid = \|\tilde{d}(n+1\colon m-p)\|_2$

Computing $\widetilde{R}$ requires $3n^2p + p^2(m/3 - p)$ flops, versus $2n^2(m-p-n/3)$ for the Householder QR factorization of $\widetilde{A}$. Note the following algorithm to compute $\widetilde{Q}$ is more economical than calling Algorithm 4.2.2 p times, saving

$3mp^2$ flops by not updating the first p rows of $\widetilde{Q}$.

Algorithm 4.2.4 *Given matrices C and S from Algorithm 4.2.3 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{(m-p)\times(m-p)}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with the kth to $(k+p-1)$st rows deleted.*

if $k \neq 1$
    % Permute Q
    $Q(p+1\colon k+p-1, 1\colon m) = Q(1\colon k-1, 1\colon m)$
end
for $i = 1\colon p$
    for $j = m-1\colon -1\colon i+1$

$$Q(p+1\colon m, j\colon j+1) = Q(p+1\colon m, j\colon j+1)\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}$$

    end
end
% Do not need to update columns $1\colon p$ of Q
$Q(p+1\colon m, i+1) = S(i,i)Q(p+1\colon m, i) + C(i,i)Q(p+1\colon m, i+1)$
$\widetilde{Q} = Q(p+1\colon m, p+1\colon m)$

Updating the QR Factorization for any m and n

The relevant parts of Algorithm 4.2.3 and Algorithm 4.2.4 could be used to update the QR factorization of $\widetilde{A}$ in the case when $m < n$ without any alteration.

4.2.2 Alternative Methods for Deleting Rows

Hyperbolic Transformations

If we have the QR factorization $A = QR \in \mathbb{R}^{m\times n}$, then

$$A^TA = R^TQ^TQR = R^TR,$$

which is a Cholesky factorization of A^TA. And if we define a permutation matrix P such that

$$PA = \begin{bmatrix} a_k^T \\ A(1\colon k-1, 1\colon n) \\ A(k+1\colon m, 1\colon n) \end{bmatrix} = \begin{bmatrix} a^T \\ \widetilde{A} \end{bmatrix} = PQR,$$

where a_k^T is the kth row of A, then we have

$$R^TQ^TP^TPQR = R^TR = A^TA = \begin{bmatrix} a_k^T \\ \widetilde{A} \end{bmatrix}^T \begin{bmatrix} a_k^T \\ \widetilde{A} \end{bmatrix}. \tag{4.2.2}$$

Thus if we find $\widetilde{R}$, such that $\widetilde{R}^T\widetilde{R} = \widetilde{A}^T\widetilde{A}$, then we have computed $\widetilde{R}$ for $\widetilde{A}$ being A with the kth row deleted. This can be achieved with *hyperbolic transformations*.

We define $W \in \mathbb{R}^{m\times m}$ as *pseudo-orthogonal* with respect to the *signature matrix*

$$J = \operatorname{diag}(\pm 1) \in \mathbb{R}^{m\times m}$$

if

$$W^TJW = J.$$

If we transform a matrix with W we say that this is a hyperbolic transformation.

Now from (4.2.2) we have

$$\begin{aligned} \widetilde{A}^T\widetilde{A} &= A^TA - a_ka_k^T \\ &= R^TR - a_ka_k^T \\ &= \begin{bmatrix} R^T & a_k \end{bmatrix} \begin{bmatrix} I_n & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} R \\ a_k^T \end{bmatrix}, \end{aligned}$$

with the signature matrix

$$J = \begin{bmatrix} I_n & 0 \\ 0 & -1 \end{bmatrix}. \tag{4.2.3}$$

And suppose there is a $W \in \mathbb{R}^{(n+1)\times(n+1)}$ such that $W^T J W = J$ with the property

$$W \begin{bmatrix} R \\ a_k^T \end{bmatrix} = \begin{bmatrix} \widetilde{R} \\ 0 \end{bmatrix}$$

is upper trapezoidal. It follows that

$$\begin{aligned} \widetilde{A}^T \widetilde{A} &= [\, R^T \quad a_k \,] W^T J W \begin{bmatrix} R \\ a_k^T \end{bmatrix} \\ &= [\, \widetilde{R}^T \quad 0 \,] J \begin{bmatrix} \widetilde{R} \\ 0 \end{bmatrix} \\ &= \widetilde{R}^T \widetilde{R}, \end{aligned}$$

which is the Cholesky factorization we seek.

We construct the hyperbolic transformation matrix, W, by a product of *hyperbolic rotations*, $W(i, n+1) \in \mathbb{R}^{(n+1)\times(n+1)}$, which are of the form

$$W(i,n+1) = \begin{array}{c} \\ i \\ \\ \\ n+1 \end{array} \begin{array}{c} \begin{array}{ccccc} i & & & & n+1 \end{array} \\ \begin{bmatrix} I & & & & \\ & c & & -s & \\ & & I & & \\ & -s & & c & \\ & & & & \end{bmatrix} \end{array}$$

where $c = \cosh(\theta)$ and $s = \sinh(\theta)$ for some θ and $c^2 - s^2 = 1$. $W(i, n+1)^T J W(i, n+1) = J$, where J is given in (4.2.3).

$W(i, n+1)x$ only transforms the ith and $(n+1)$st elements. To solve the 2×2 problem

$$\begin{bmatrix} c & -s \\ -s & c \end{bmatrix} \begin{bmatrix} x_i \\ x_{n+1} \end{bmatrix} = \begin{bmatrix} y \\ 0 \end{bmatrix},$$

we note that $cx_{n+1} = sx_i$ and there is no solution for $x_i = x_{n+1} \neq 0$. If $x_i \neq x_{n+1}$ then we can compute c and s with the following algorithm.

Algorithm 4.2.5 *This algorithm generates scalars c and s such that* $\begin{bmatrix} c & -s \\ -s & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y \\ 0 \end{bmatrix}$ *where x_1, x_2 and y are scalars and $c^2 - s^2 = 1$, if a solution exists.*

```
if x_2 = 0
    s = 0
    c = 1
else
    if |x_2| < |x_1|
        t = x_2/x_1
        c = 1/sqrt(1 - t^2)
        s = ct
    else
        no solution exists
    end
end
```

Note the norm of the rotation gets large as x_1 gets close to x_2.

We thus generate n hyperbolic transformations such that

$$W(n, n+1) \ldots W(2, n+1) W(1, n+1) \begin{bmatrix} R \\ a_k^T \end{bmatrix} = \begin{bmatrix} \widetilde{R} \\ 0 \end{bmatrix}.$$

It turns out that all the $W(i, n+1)$ can be found if A has full rank [4].

Chamber's Algorithm

A method due to Chambers [12] mixes a hyperbolic and Givens rotation. If we have our usual Givens transformation on the vector x

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

then the transformed x_i, $\tilde{x}_i$, are

$$\begin{aligned} \tilde{x}_1 &= cx_1 - sx_2, \\ \tilde{x}_2 &= sx_1 + cx_2, \end{aligned} \tag{4.2.4}$$

with

$$c = \frac{x_1}{\sqrt{x_1^2 + x_2^2}}, \quad s = \frac{-x_2}{\sqrt{x_1^2 + x_2^2}}.$$

Now suppose we know $\tilde{x}_1$ and want to recreate the vector x, then rearranging (4.2.4) we have

$$\begin{aligned} x_1 &= (sx_2 + \tilde{x}_1)/c, \\ \tilde{x}_2 &= sx_1 + cx_2, \end{aligned}$$

with

$$c = \frac{\sqrt{\tilde{x}_1^2 - x_2^2}}{\tilde{x}_1}, \quad s = \frac{-x_2}{\tilde{x}_1}.$$

Thus we can recreate the steps that would have updated $\widetilde{R}$ had we *added* a_k^T to $\widetilde{A}$, instead of deleting it from A. At the ith step, for $i = 1\colon n+1$, with

$$\begin{aligned} x_1 &= R(i,i), \\ \tilde{x}_1 &= \widetilde{R}(i,i), \\ x_2 &= a_k^{(i-1)}(i), \end{aligned}$$

we compute, for $j = i\colon n+1$:

$$\begin{aligned} \widetilde{R}(i,j) &= (R(i,i) + sa_k(j))/c, \\ a_k^{(i)}(j) &= sR(i,j) + ca_k^{(i-1)}(j). \end{aligned}$$

Saunders' Algorithm

If Q is not available then Saunders' algorithm [44] offers an alternative to Algorithm 4.2.1. The first row of

$$PA = \begin{bmatrix} a_k^T \\ \widetilde{A} \end{bmatrix} = PQ \begin{bmatrix} R_1 \\ 0 \end{bmatrix},$$

can be written

$$a_k^T = q^T \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = [\, q_1^T q_2^T \,] \begin{bmatrix} R_1 \\ 0 \end{bmatrix},$$

where $q_1 \in \mathbb{R}^n$. We compute q_1 by solving

$$R_1^T q_1 = a_k,$$

and since $\|q\|_2 = 1$ we have

$$\eta = \|q_2\|_2 = (1 - \|q_1\|_2^2)^{1/2}.$$

Then we have, with the same Givens matrices in (4.2.1),

$$G(n+1, n+2)^T \ldots G(m-1, m)^T \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} q_1 \\ \pm\eta \\ 0 \end{bmatrix},$$

which would not effect R. So we need only compute

$$G(1, 2)^T \ldots G(n, n+1)^T \begin{bmatrix} q_1 \\ \eta \end{bmatrix} = \alpha e_1, \quad |\alpha| = 1,$$

and update R by

$$G(1, 2)^T \ldots G(n, n+1)^T R = \begin{bmatrix} v^T \\ \widetilde{R} \end{bmatrix}.$$

This algorithm is implemented in LINPACK's `xCHDD`.

Stability Issues

Stewart [49] shows that hyperbolic transformations are not backward stable. However, Chamber's and Saunder's algorithms are *relationally stable* [6], [47], that is if W represents the product of all the transformations then

$$W^T R = \begin{bmatrix} v^T \\ \widetilde{R} \end{bmatrix} + E,$$

where

$$\|E\| \leq c_n u \|R\|,$$

and c_n is a constant that depends on n.

Saunder's algorithm can fail for certain data, see [5].

Block Downdating

Hyperbolic transformations have been generalized by Rader and Steinhardt as *hyperbolic Householder transformations* [42].

Alternatives are discussed in Elden and Park [22], and the references contained within, including a generalization of Saunders Algorithm. See also Bojanczyk, Higham and Patel [7] and Olskanskyj, Lebak and Bojanczyk [39].

4.2.3 Adding Rows

Adding One Row

If we wish to *add* an *observation* to our least squares problem then we need to add a row, $u^T \in \mathbb{R}^n$, in the kth position, $k = 1\!:\!m+1$, of $A = QR \in \mathbb{R}^{m\times n}$, $m \geq n$. We can then write

$$\widetilde{A} = \begin{bmatrix} A(1\!:\!k-1, 1\!:\!n) \\ u^T \\ A(k\!:\!m, 1\!:\!n) \end{bmatrix}$$

and we can define a permutation matrix, P, such that

$$P\widetilde{A} = \begin{bmatrix} A \\ u^T \end{bmatrix},$$

and then

$$\begin{bmatrix} Q^T & 0 \\ 0 & 1 \end{bmatrix} P\widetilde{A} = \begin{bmatrix} R \\ u^T \end{bmatrix}. \tag{4.2.5}$$

For example, with $m = 8$ and $n = 6$ the right-hand side of (4.2.5) looks like:

$$\begin{bmatrix} + & + & + & + & + & + \\ 0 & + & + & + & + & + \\ 0 & 0 & + & + & + & + \\ 0 & 0 & 0 & + & + & + \\ 0 & 0 & 0 & 0 & + & + \\ 0 & 0 & 0 & 0 & 0 & + \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \ominus & \ominus & \ominus & \ominus & \ominus & \ominus \end{bmatrix},$$

with the nonzero elements of R represented with a $+$ and the elements to be eliminated are shown with a $\ominus$.

Thus to find $\widetilde{A} = \widetilde{Q}\widetilde{R}$, we can define n Givens matrices, $G(i,j) \in \mathbb{R}^{m+1 \times m+1}$, to eliminate $u^T$to give

$$G(n, m+1)^T \ldots G(1, m+1)^T \begin{bmatrix} R \\ u^T \end{bmatrix} = \widetilde{R},$$

so we have

$$\widetilde{A} = \left(P^T \begin{bmatrix} Q & 0 \\ 0 & 1 \end{bmatrix} G(1, m+1) \ldots G(n, m+1) \right) \widetilde{R} = \widetilde{Q}\widetilde{R}.$$

and to update b we compute

$$G(n, m+1)^T \ldots G(1, m+1)^T \begin{bmatrix} Q^T b \\ \mu \end{bmatrix} = \tilde{d},$$

where μ is the element inserted into b corresponding to u^T. This gives the following algorithm.

Algorithm 4.2.6 *Given $A = QR \in \mathbb{R}^{m \times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T \widetilde{A} = \widetilde{R} \in \mathbb{R}^{(m+1) \times n}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with a row, $u^T \in \mathbb{R}^n$, inserted in the kth position, $1 \leq k \leq m+1$, and $\tilde{d}$ such that $\|\widetilde{A}x - \tilde{b}\|_2 = \|\widetilde{R}x - \tilde{d}\|_2$, where $\tilde{b}$ is b with a scalar μ inserted in the kth position. The residual, $\|\tilde{d}(n+1\colon m+1)\|_2$, is also computed.*

$d = Q^T b$
for $j = 1{:}\, n$
    $[c(j), s(j)] = \mathbf{givens}(R(j,j), u(j))$
    $R(j,j) = c(j)R(j,j) - s(j)u(j)$
    % Update jth row of R and u
    $t1 = R(j, j+1{:}\, n)$
    $t2 = u(j+1{:}\, n)$
    $R(j, j+1{:}\, n) = c(j)t1 - s(j)t2$
    $u(j+1{:}\, n) = s(j)t1 + c(j)t2$
    % Update jth row of d and μ
    $t1 = d(j)$
    $t2 = \mu$
    $d(j) = c(j)t1 - s(j)t2$
    $\mu = s(j)t1 + c(j)t2$
end
$\widetilde{R} = \begin{bmatrix} R \\ 0 \end{bmatrix}$
$\tilde{d} = \begin{bmatrix} d \\ \mu \end{bmatrix}$
% Compute the residual
$resid = \|\tilde{d}(n+1{:}\, m+1)\|_2$

Computing $\widetilde{R}$ requires $3n^2$ flops, versus $2n^2(m - n/3)$ for the Householder QR factorization of $\widetilde{A}$. If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.7 *Given vectors c and s from Algorithm 4.2.6 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{(m+1)\times(m+1)}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with a row added in the kth position.*

Set $\widetilde{Q} = \begin{bmatrix} Q & 0 \\ 0 & 1 \end{bmatrix}$
if $k \neq m+1$
    % Permute Q
    $Q = \begin{bmatrix} Q(1{:}\, k-1, 1{:}\, n) \\ Q(m+1, 1{:}\, n) \\ Q(k{:}\, m, 1{:}\, n) \end{bmatrix}$
end
for $j = 1{:}\, n$

$t1 = \widetilde{Q}(1\colon m+1, j)$
$t2 = \widetilde{Q}(1\colon m+1, m+1)$
$\widetilde{Q}(1\colon m+1, j) = c(j)t1 - s(j)t2$
$\widetilde{Q}(1\colon m+1, m+1) = s(j)t1 + c(j)t2$
end

Adding a Block of Rows

To add a block of p observations to our least squares problem we add a block of p rows, $U \in \mathbb{R}^{(p\times n)}$, in the kth to $(k+p-1)$st positions, $k = 1\colon m+1$, of $A = QR \in \mathbb{R}^{m\times n}$, $m \geq n$, we can then write

$$\widetilde{A} = \begin{bmatrix} A(1\colon k-1, 1\colon n) \\ U \\ A(k\colon m, 1\colon n) \end{bmatrix}$$

and we can define a permutation matrix, P, such that

$$P\widetilde{A} = \begin{bmatrix} A \\ U \end{bmatrix},$$

and

$$\begin{bmatrix} Q^T & 0 \\ 0 & I_p \end{bmatrix} P\widetilde{A} = \begin{bmatrix} R \\ U \end{bmatrix}. \tag{4.2.6}$$

For example, with $m = 8$, $n = 6$ and $p = 3$ the right-hand side of Equation (4.2.6) looks like:

$$\begin{bmatrix} + & + & + & + & + & + \\ 0 & + & + & + & + & + \\ 0 & 0 & + & + & + & + \\ 0 & 0 & 0 & + & + & + \\ 0 & 0 & 0 & 0 & + & + \\ 0 & 0 & 0 & 0 & 0 & + \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \ominus & \ominus & \ominus & \ominus & \ominus & \ominus \\ \ominus & \ominus & \ominus & \ominus & \ominus & \ominus \\ \ominus & \ominus & \ominus & \ominus & \ominus & \ominus \end{bmatrix},$$

with the nonzero elements of R represented with a $+$ and the elements to be eliminated are shown with a $\ominus$.

Thus to find $\widetilde{A} = \widetilde{Q}\widetilde{R}$, we can define n Householder matrices to eliminate U to give

$$H_n \ldots H_1 \begin{bmatrix} R \\ U \end{bmatrix} = \widetilde{R},$$

so we have

$$\widetilde{A} = \left(P^T \begin{bmatrix} Q & 0 \\ 0 & I_p \end{bmatrix} H_1 \ldots H_n \right) \widetilde{R} = \widetilde{Q}\widetilde{R}.$$

The Householder matrix, $H_j \in \mathbb{R}^{(m+p)\times(m+p)}$, will zero the jth column of U. Its associated Householder vector, $v_j \in \mathbb{R}^{(m+p)}$, is such that

$$\left.\begin{array}{rcl} v_j(1\colon j-1) &=& 0, \\ v_j(j) &=& 1, \\ v_j(j+1\colon m) &=& 0, \\ v_j(m+1\colon m+p) &=& x/(r_{jj} - \| \begin{bmatrix} r_{jj} & x^T \end{bmatrix} \|_2), \text{ where } x = U(1\colon p, j). \end{array}\right\} \tag{4.2.7}$$

So the H_j have the following structure

$$H_j = \begin{bmatrix} I & & & \\ & h_{jj} & & \begin{bmatrix} h_{j,m+1} & \ldots & h_{j,m+p} \end{bmatrix} \\ & & I & \\ & \begin{bmatrix} h_{m+1,j} \\ \vdots \\ h_{m+p,j} \end{bmatrix} & & \begin{bmatrix} h_{m+1,m+1} & \ldots & h_{m+1,m+p} \\ \vdots & & \vdots \\ h_{m+p,m+1} & \ldots & h_{m+p,m+p} \end{bmatrix} \end{bmatrix}.$$

Then to update b we compute

$$H_n \ldots H_1 \begin{bmatrix} Q^T b \\ e \end{bmatrix} = \tilde{d},$$

where e is such that $Ux = e$. This gives the following algorithm.

Algorithm 4.2.8 *Given $A = QR \in \mathbb{R}^{m\times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{(m+p)\times n}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with a block of rows, $U \in \mathbb{R}^{p\times n}$, inserted in the kth to $(k+p-1)$st positions, $1 \leq k \leq m+1$, $p \geq 1$, and $\tilde{d}$ such that $\|\widetilde{A}x - \tilde{b}\|_2 = \|\widetilde{R}x - \tilde{d}\|_2$, where $\tilde{b}$ is b with the vector e inserted in the kth to $(k+p-1)$st positions. The residual, $\|\tilde{d}(n+1\colon m+p)\|_2$, is also computed.*

$d = Q^T b$
for $j = 1\colon n$
  $[V(1\colon p, j), \tau(j)] = \mathbf{householder}(R(j,j), U(1\colon p, j))$
  % Remember old jth row of R
  $R_j = R(j, j+1\colon n)$
  % Update jth row of R
  $R(j, j\colon n) = (1-\tau(j))R(j, j\colon n) - \tau(j)V(1\colon p, j)^T U(1\colon p, j\colon n)$
  % Update trailing part if U
  if $j < n$
    $U(1\colon p, j+1\colon n) = U(1\colon p, j+1\colon n) - \tau(j)V(1\colon p, j)R_j$
      $-\tau(j)V(1\colon p, j)(V(1\colon p, j)^T U(1\colon p, j+1\colon n))$
  end
  % Remember old jth element of d
  $d_j = R(j)$
  % Update jth element of d
  $d(j) = (1-\tau(j))d(j) - \tau(j)V(1\colon p, j)^T e(1\colon p)$
  % Update e
  $e(1\colon p) = e(1\colon p) - \tau(j)V(1\colon p, j)d_j$
      $-\tau(j)V(1\colon p, j)(V(1\colon p, j)^T e(1\colon p))$
end
$\widetilde{R} = \begin{bmatrix} R \\ 0 \end{bmatrix}$
$\tilde{d}\begin{bmatrix} d \\ e \end{bmatrix}$
% Compute the residual
$resid = \|\tilde{d}(n+1\colon m+p)\|_2$

Computing $\widetilde{R}$ requires $2n^2p$ flops, versus $2n^2(m+p-n/3)$ for the Householder QR factorization of $\widetilde{A}$. If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.9 *Given the matrix V and vector τ from Algorithm 4.2.8 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{(m+p)\times(m+p)}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$,*

where $\widetilde{A}$ is the matrix $A = QR$ with a block of rows inserted in the kth to $(k+p-1)$st positions.

Set $\widetilde{Q} = \begin{bmatrix} Q & 0 \\ 0 & I \end{bmatrix}$
if $k \neq m+1$
    % Permute Q
    $\widetilde{Q} = \begin{bmatrix} \widetilde{Q}(1\colon k-1, 1\colon m+p) \\ \widetilde{Q}(m+1\colon m+p, 1\colon m+p) \\ \widetilde{Q}(k\colon m, 1\colon m+p) \end{bmatrix}$
end
for $j = 1\colon n$
    % Remember jth column of $\widetilde{Q}$
    $\widetilde{Q}_k = \widetilde{Q}(1\colon m+p, j)$
    % Update jth column
    $Q(1\colon m+p, j) = Q(1\colon m+p, j)(1-\tau(j)) -$
        $\widetilde{Q}(1\colon m+p, m+1\colon m+p)\tau(j)V(1\colon p, j)$
    % Update $m+1\colon p$ columns of $\widetilde{Q}$
    $\widetilde{Q}(1\colon m+p, m+1\colon m+p) = \widetilde{Q}(1\colon m+p, m+1\colon m+p)$
        $-\tau(j)\widetilde{Q}_k V(1\colon p, j)^T$
        $-\tau(j)(\widetilde{Q}(1\colon m+p, m+1\colon m+p)V(1\colon p, j))V(1\colon p, j)^T$
end

This algorithm could be made more economical by noting that at the jth stage, for $i > m$, $\tilde{q}_{ij} = 0$, and avoiding some unnecessary multiplications by zero. Also $\widetilde{Q}(m+1\colon m+p, 1\colon m) = 0$ and $\widetilde{Q}(m+1\colon m+p, m+1\colon m+p) = I_p$, prior to the permutation.

Algorithm 4.2.8 can be improved by exploiting the Level 3 BLAS by using the representation of the product of n_b Householder matrices, H_i, as

$$H_1 H_2 \ldots H_{n_b} = I - VTV^T, \tag{4.2.8}$$

where

$$V = [\, v_1 \quad v_2 \quad \ldots \quad v_{n_b} \,] = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix},$$

and $V_1 \in \mathbb{R}^{n_b \times n_b}$ is lower triangular and T is upper triangular. We can write

$$Q^T \widetilde{A} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \\ 0 & 0 & 0 \\ U_{11} & U_{12} & U_{13} \end{bmatrix}, \tag{4.2.9}$$

where the R_{ii} are upper triangular with $R_{11} \in \mathbb{R}^{r \times r}$ and $R_{22} \in \mathbb{R}^{n_b \times n_b}$, and after we have updated the first r columns, then the transformed right-hand side of (4.2.9) looks like:

$$\begin{bmatrix} R_{11}^{(r)} & R_{12}^{(r)} & R_{13}^{(r)} \\ 0 & R_{22}^{(r)} & R_{23}^{(r)} \\ 0 & 0 & R_{33}^{(r)} \\ 0 & 0 & 0 \\ 0 & U_{12}^{(r)} & U_{13}^{(r)} \end{bmatrix}.$$

Now we eliminate the first column of $U_{12}^{(r)}$ and instead of updating the trailing parts of R and U we update only the trailing parts of $U_{12}^{(r)}$ and the $(r+1)$st row of $R_{22}^{(r)}$, which are the only elements affected in this middle block column, and continue in this way until U_{12} has been eliminated. We can then employ the representation (4.2.8) to apply n_b Householder matrices to update the last block column in one go. We have, by the definition of the Householder vectors in (4.2.7)

$$V = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}, \quad V_1 = I_{n_b}, \quad V_2 = \begin{bmatrix} 0 \\ \overline{V}_2 \end{bmatrix},$$

where $\overline{V}_2 \in \mathbb{R}^{p \times n_b}$ hold the essential part of the Householder vectors for the current block column. Then

$$[\, I - VT^TV^T \,]^T \begin{bmatrix} R_{23} \\ R_{33} \\ 0 \\ U_{13} \end{bmatrix} = \begin{bmatrix} I_{m+p-r} - \begin{bmatrix} I_{n_b} \\ 0 \\ 0 \\ \overline{V}_2 \end{bmatrix} T^T \begin{bmatrix} I_{n_b} & 0 & 0 & \overline{V}_2^T \end{bmatrix} \end{bmatrix} \begin{bmatrix} R_{23} \\ R_{33} \\ 0 \\ U_{13} \end{bmatrix}$$

$$= \begin{bmatrix} (I_{n_b} - T^T)R_{23} - T^T\overline{V}_2^T U_{13} \\ R_{33} \\ 0 \\ -\overline{V}_2 T^T R_{23} + (I - \overline{V}_2 T^T \overline{V}_2^T)U_{13} \end{bmatrix},$$

This approach leads to a blocked algorithm, where at the kth stage we factorize $[\, R_{22}^T \quad 0 \quad U_{12}^T \,]^T$, where $R_{22} \in \mathbb{R}^{n_b \times n_b}$ and $U_{12} \in \mathbb{R}^{p \times n_b}$, then update $R_{23} \in \mathbb{R}^{n_b \times (n-kn_b)}$ and $U_{13}^T \in \mathbb{R}^{p \times (n-kn_b)}$ as above. And to update $Q^T b = d$ we compute

$$\begin{bmatrix} d(1\colon (k-1)n_b) \\ (I_{n_b} - T^T)d((k-1)n_b + 1\colon kn_b) - T^T\overline{V}_2^T e \\ d(kn_b + 1\colon m) \\ -\overline{V}_2 T^T d((k-1)n_b + 1\colon kn_b) + (I - \overline{V}_2 T^T \overline{V}_2^T)e \end{bmatrix} = \begin{bmatrix} d \\ e \end{bmatrix}.$$

Algorithm 4.2.10 *Given $A = QR \in \mathbb{R}^{m \times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T \widetilde{A} = \widetilde{R} \in \mathbb{R}^{(m+p) \times n}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with a block of rows, $U \in \mathbb{R}^{p \times n}$, inserted in the kth to $(k+p-1)$st positions, $1 \leq k \leq m+1$, $p \geq 1$, and $\tilde{d}$ such that $\|\widetilde{A}x - \tilde{b}\|_2 = \|\widetilde{R}x - \tilde{d}\|_2$, where $\tilde{b}$ is b with the vector e inserted in the kth to $(k+p-1)$st positions. The residual, $\|\tilde{d}(n+1\colon m+p)\|_2$, is also computed. This is a Level 3 BLAS algorithm with block size n_b.*

```
d = Q^T b
for k = 1:n_b:n
    % Check for the last column block
    jb = min(n_b, n - k + 1)
    Factorize current block with Algorithm 4.2.8 where
    V is V(1:p, k:k + jb - 1)
    % If we are not in last block column build T
    % and update trailing matrix
    if k + jb ≤ n
        for j = k:k + jb - 1
            % Build T
            if j = k
                T(1,1) = τ(j)
            else
```

$T(1\colon j-k, j-k+1) = -\tau(j)T(1\colon j-k, 1\colon j-k)$
$*V(1\colon p, k\colon j-1)^T V(1\colon p, j)$
$T(j-k+1, j-k+1) = \tau(j)$
end
end
% Compute products we use more than once
$T_V = T^T V(1\colon p, k\colon k+jb-1)^T$
$T_e = T_V e$
$T_U = T_V U(1\colon p, , k+jb\colon n)$
% Remember old d and e
$d_k = d(k\colon k+jb-1)$
$e_k = e$
% Update d and e
$d(k\colon k+jb-1) = d_k - T^T d_k - T_e$
$e = -V(1\colon p, k\colon k+jb-1)T^T d_k + e_k$
$-V(1\colon p, k\colon k+jb-1)T_e$
% Remember old trailing parts of R and U
$R_k = R(k\colon k+jb-1, k+jb\colon n)$
$U_k = U(1\colon p, k+jb\colon n)$
% Update trailing parts of R and U
$R(k\colon k+jb-1, k+jb\colon n) = R_k - T^T R_k - T_U$
$U(1\colon p, k+jb\colon n) = -V(1\colon p, k\colon k+jb-1)T^T R_k + U_k$
$-V(1\colon p, k\colon k+jb-1)T_U$
end
end
$\widetilde{R} = \begin{bmatrix} R \\ 0 \end{bmatrix}$
$\tilde{d}\begin{bmatrix} d \\ e \end{bmatrix}$
% Compute the residual
$resid = \|\tilde{d}(n+1\colon m+p)\|_2$

We could apply the same approach to improve Algorithm 4.2.9.

Updating the QR Factorization for any m and n

In the case where $m < n$ after m steps of Algorithm 4.2.8 we have

$$\widetilde{A} = P^T \begin{bmatrix} Q^T & 0 \\ 0 & I_p \end{bmatrix} H_1 \ldots H_n \begin{bmatrix} R_{11} & R_{22} \\ 0 & V \end{bmatrix},$$

where R_{11} is upper triangular and V is the transformed $U(1\colon p, m+1\colon n)$. Thus if we compute the QR factorization $V = Q_V R_V$, we than have

$$\widetilde{A} = \left(P^T \begin{bmatrix} Q^T & 0 \\ 0 & I_p \end{bmatrix} H_1 \dots H_n \begin{bmatrix} I_m & 0 \\ 0 & Q_V^T \end{bmatrix} \right) \widetilde{R} = \widetilde{Q}\widetilde{R}.$$

This gives us the following algorithms to update the QR factorization for any m and n.

Algorithm 4.2.11 *Given $A = QR \in \mathbb{R}^{m\times n}$ this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{(m+p)\times n}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with a block of rows, $U \in \mathbb{R}^{p\times n}$, inserted in the kth to $(k+p-1)$st positions, $1 \le k \le m+1$, $p \ge 1$.*

$lim = \min(m, n)$
for $j = 1\colon lim$
  $[V(1\colon p, j), \tau(j)] =$ **householder**$(R(j, j), U(1\colon p, j))$
  % Remember old jth row of R
  $R_k = R(j, j+1\colon n)$
  % Update jth row of R
  $R(j, j\colon n) = (1-\tau(j))R(j, j\colon n) - \tau(j)V(1\colon p, j)^T U(1\colon p, j\colon n)$
  % Update trailing part if U
  if $j < n$
    $U(1\colon p, j+1\colon n) = U(1\colon p, j+1\colon n) - \tau(j)V(1\colon p, j)R_k$
    $\quad -\tau(j)V(1\colon p, j)(V(1\colon p, j)^T U(1\colon p, j+1\colon n))$
  end
end
$\widetilde{R} = \begin{bmatrix} R \\ 0 \end{bmatrix}$
if $m < n$
  Perform the QR factorization $U(:, m+1\colon n) = Q_U R_U$
  $\widetilde{R}(m+1\colon m+p, m+1\colon n) = R_U$
end

This algorithm could also be improved by using the representation (4.2.8) to include a Level 3 BLAS part. If $\widetilde{Q}$ is required it can be computed with the following algorithm.

Algorithm 4.2.12 *Given the matrices V and Q_U and vector τ from Algorithm 4.2.11 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{(m+p)\times(m+p)}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with a block of rows inserted in the kth to $(k+p-1)$st positions.*

Set $\widetilde{Q} = \begin{bmatrix} Q & 0 \\ 0 & I \end{bmatrix}$
if $k \neq m+1$
    % Permute Q
    $\widetilde{Q} = \begin{bmatrix} \widetilde{Q}(1\colon k-1, 1\colon m+p) \\ \widetilde{Q}(m+1\colon m+p, 1\colon m+p) \\ \widetilde{Q}(k\colon m, 1\colon m+p) \end{bmatrix}$
end
$lim = \min(m, n)$
for $j = 1\colon lim$
    % Remember jth column of $\widetilde{Q}$
    $\widetilde{Q}_k = \widetilde{Q}(1\colon m+p, j)$
    % Update jth column
    $Q(1\colon m+p, j) = Q(1\colon m+p, j)(1-\tau(j))$
        $-\widetilde{Q}(1\colon m+p, m+1\colon m+p)\tau(j)V(1\colon p, j)$
    % Update $m+1\colon p$ columns of $\widetilde{Q}$
    $\widetilde{Q}(1\colon m+p, m+1\colon m+p) = \widetilde{Q}(1\colon m+p, m+1\colon m+p)$
        $-\tau(j)\widetilde{Q}_k V(1\colon p, j)^T$
        $-\tau(j)(\widetilde{Q}(1\colon m+p, m+1\colon m+p)V(1\colon p, j))V(1\colon p, j)^T$
end
if $m < n$
    $Q(1\colon m+p, m+1\colon m+p) = Q(1\colon m+p, m+1\colon m+p)Q_U$
end

4.2.4 Deleting Columns

Deleting One Column

If we wish to *delete* a *variable* from our least squares problem then we have the problem of updating the QR factorization of A where we delete the kth column, $k \neq n$, from A, we can write

$$\widetilde{A} = [\, A(1\colon m, 1\colon k-1) \quad A(1\colon m, k+1\colon n)\,]$$

then

$$Q^T\widetilde{A} = [\, R(1\colon m, 1\colon k-1) \quad R(1\colon m, k+1\colon n)\,]\,. \tag{4.2.10}$$

For example, with $m = 8$, $n = 6$ and $k = 3$ the right-hand side of Equation (4.2.10) looks like:

$$\begin{bmatrix} + & + & + & + & + \\ 0 & + & + & + & + \\ 0 & 0 & + & + & + \\ 0 & 0 & \ominus & + & + \\ 0 & 0 & 0 & \ominus & + \\ 0 & 0 & 0 & 0 & \ominus \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

with the nonzero elements to remain represented with a $+$ and the elements to be eliminated are shown with a $\ominus$.

Thus we can define $n-k$ Givens matrices, $G(i,j) \in \mathbb{R}^{m\times m}$, to eliminate the subdiagonal elements of $Q^T\widetilde{A}$ to give

$$(G(n,n+1)^T \ldots G(k,k+1)^T Q^T)\widetilde{A} = \widetilde{Q}^T\widetilde{A} = \widetilde{R},$$

where $\widetilde{R} \in \mathbb{R}^{m\times(n-1)}$ is upper trapezoidal and $\widetilde{Q} \in \mathbb{R}^{m\times m}$ is orthogonal, and to update b we compute

$$G(n,n+1)^T \ldots G(k,k+1)^T Q^T b = \tilde{d}.$$

This gives the following algorithm.

Algorithm 4.2.13 *Given $A = QR \in \mathbb{R}^{m\times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{m\times(n-1)}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with the kth column deleted, $1 \leq k \leq n-1$, and $\tilde{d}$ such that $\|\widetilde{A}x - b\|_2 = \|\widetilde{R}x - \tilde{d}\|_2$. The residual, $\|\tilde{d}(n+1\colon m)\|_2$, is also computed.*

$\tilde{d} = Q^T b$
set $R(1\colon m, k\colon n-1) = R(1\colon m, k+1\colon n)$

for $j = k\colon n-1$
$[c(j), s(j)] = \mathbf{givens}(R(j,j), R(j+1,j))$
% Update R
$R(j,j) = c(j)R(j,j) - s(j)R(j+1,j)$
$R(j\colon j+1, j+1\colon n-1) = \begin{bmatrix} c(j) & s(j) \\ -s(j) & c(j) \end{bmatrix}^T R(j\colon j+1, j+1\colon n-1)$
% Update $\tilde{d}$
$\tilde{d}(j\colon j+1) = \begin{bmatrix} c(j) & s(j) \\ -s(j) & c(j) \end{bmatrix}^T \tilde{d}(j\colon j+1)$
end
$\widetilde{R} =$ upper triangular part of $R(1\colon m, 1\colon n-1)$
% Compute the residual
$resid = \|\tilde{d}(n+1\colon m)\|_2$

Computing $\widetilde{R}$ requires $n^2/2 - nk + k^2/2$ flops, versus $2n^2(m - n/3)$ for the Householder QR factorization of $\widetilde{A}$. If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.14 *Given vectors c and s from Algorithm 4.2.13 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{m\times m}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with the kth column deleted.*

for $j = k\colon n-1$
$Q(1\colon m, j\colon j+1) = Q(1\colon m, j\colon j+1) \begin{bmatrix} c(j) & s(j) \\ -s(j) & c(j) \end{bmatrix}$
end
$\widetilde{Q} = Q$

In the case when $k = n$ then

$$\widetilde{A} = A(1\colon m, 1\colon k-1), \quad \widetilde{R} = R(1\colon m, 1\colon k-1), \quad \widetilde{Q} = Q, \text{ and } \tilde{d} = Q^T b,$$

and there is no computation to do.

Deleting a Block of Columns

To delete a block of p variables from our least squares problem we delete a block of p columns, from the kth column onwards, from A and we can write

$$\widetilde{A} = [\, A(1\colon m, 1\colon k-1) \quad A(1\colon m, k+p\colon n) \,]$$

then

$$Q^T \widetilde{A} = [\, R(1\colon m, 1\colon k-1) \quad R(1\colon m, k+p\colon n) \,]\,. \tag{4.2.11}$$

For example, with $m = 10$, $n = 8$, $k = 3$ and $p = 3$ the right-hand side of Equation (4.2.11) looks like:

$$\begin{bmatrix} + & + & + & + & + \\ 0 & + & + & + & + \\ 0 & 0 & + & + & + \\ 0 & 0 & \ominus & + & + \\ 0 & 0 & \ominus & \ominus & + \\ 0 & 0 & \ominus & \ominus & \ominus \\ 0 & 0 & 0 & \ominus & \ominus \\ 0 & 0 & 0 & 0 & \ominus \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

with the nonzero elements to remain represented with a $+$ and the elements to be eliminated are shown with a $\ominus$.

Thus we can define $n - p - k + 1$ Householder matrices, $H_j \in \mathbb{R}^{m \times m}$, with associated Householder vectors, $v_j \in \mathbb{R}^{(p+1)}$ such that

$$\begin{aligned} v_j(1\colon j-1) &= 0, \\ v_j(j) &= 1, \\ v_j(j+1\colon j+p) &= x/((\widetilde{Q}^T \widetilde{A})_{jj} - \| \,[\, (\widetilde{Q}^T \widetilde{A})_{jj} \quad x^T \,]\, \|_2), \\ &\quad \text{where } x = Q^T \widetilde{A}(j+1\colon j+p, j), \\ v_j(j+p+1\colon m) &= 0. \end{aligned}$$

The H_j have the following structure

$$H_j = \begin{bmatrix} I & & & \\ & \begin{bmatrix} h_{j,j} & \cdots & h_{j,j+p} \\ \vdots & & \vdots \\ h_{j+p,j} & \cdots & h_{j+p,j+p} \end{bmatrix} & & \\ & & I & \end{bmatrix},$$

and can be used to eliminate the subdiagonal of $Q^T\widetilde{A}$ to give

$$(H_{n-p}\ldots H_kQ^T)\widetilde{A} = \widetilde{Q}^T\widetilde{A} = \widetilde{R},$$

where $\widetilde{R} \in \mathbb{R}^{m\times(n-p)}$ is upper trapezoidal and $\widetilde{Q} \in \mathbb{R}^{m\times m}$ is orthogonal, and we update b by computing

$$H_{n-p}\ldots H_kQ^Tb = \tilde{d}.$$

This gives the following algorithm.

Algorithm 4.2.15 *Given $A = QR \in \mathbb{R}^{m\times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{m\times(n-p)}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with the kth to $(k+p-1)$st columns deleted, $1 \leq k \leq n-p$, $1 \leq p < n$, and $\tilde{d}$ such that $\|\widetilde{A}x-b\|_2 = \|\widetilde{R}x-\tilde{d}\|_2$. The residual, $\|\tilde{d}(n+1\colon m)\|_2$, is also computed.*

$\tilde{d} = Q^Tb$
set $R(1\colon m, k\colon n-p) = R(1\colon m, k+p\colon n)$
for $j = k\colon n-p$
  $[V(1\colon p, j), \tau(j)] =$ **householder**$(R(j,j), R(j+1\colon j+p, j))$
  % Update R
  $R(j,j) = R(j,j) - \tau(j)R(j,j) - \tau(j)V(1\colon p, j)^TR(j+1\colon j+p, j)$
  if $j < n-p$
    $R(j\colon j+p, j+1\colon n-p) = R(j\colon j+p, j+1\colon n-p)$
    $-\tau(j)\begin{bmatrix}1\\ V(1\colon p,j)\end{bmatrix}([\,1 \quad V(1\colon p,j)^T\,]\,R(j\colon j+p, j+1\colon n-p))$
  end
  % Update $\tilde{d}$
  $\tilde{d}(j\colon j+p) = \tilde{d}(j\colon j+p, j+1)$
    $-\tau(j)\begin{bmatrix}1\\ V(1\colon p,j)\end{bmatrix}([\,1 \quad V(1\colon p,j)^T\,]\,\tilde{d}(j\colon j+p))$
end
$\widetilde{R} =$ upper triangular part of $R(1\colon m, 1\colon n-p)$
% Compute the residual
$resid = \|\tilde{d}(n+1\colon m)\|_2$

Computing $\widetilde{R}$ requires $4(np(n/2-p-k) + p^2(p/2+k) + pk^2$ flops, versus $2(n-p)^2(m-(n-p)/3)$ for the Householder QR factorization of $\widetilde{A}$. If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.16 *Given the matrix V and vector τ from Algorithm 4.2.15 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{m\times m}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with the kth to $(k+p-1)$st columns deleted.*

for $j = k\colon n-p$
$Q(1\colon m, j\colon j+p) = Q(1\colon m, j\colon j+p)$
$$-\tau(j)\Big(Q(1\colon m, j\colon j+p)\begin{bmatrix}1\\ V(1\colon p, j)\end{bmatrix}\Big)\begin{bmatrix}1 & V(1\colon p, j)^T\end{bmatrix}$$
end
$\widetilde{Q} = Q$

In the case when $k = n-p+1$ then

$$\widetilde{A} = A(1\colon m, 1\colon k-1), \quad \widetilde{R} = R(1\colon m, 1\colon k-1), \quad \widetilde{Q} = Q, \text{ and } \tilde{d} = Q^T b,$$

and there is no computation to do.

Updating the QR Factorization for any m and n

In the case when $m < n$ we need to:

- Increase the number of steps, we introduce lim, the last column to be updated.
- Determine the last index of the Householder vectors, which cannot exceed m.

This gives the following algorithms to update the QR factorization for any m and n.

Algorithm 4.2.17 *Given $A = QR \in \mathbb{R}^{m\times n}$ this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{m\times(n-p)}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with the kth to $(k+p-1)$st columns deleted, $1 \le k \le \min(m-1, n-p)$, $1 \le p < n$.*

set $R(1\colon m, k\colon n-p) = R(1\colon m, k+p\colon n)$
$lim = \min(m-1, n-p)$

for $j = k\colon lim$
$last = \min(j+p, m)$
$\quad [V(1\colon last-j, j), \tau(j)] = \mathbf{householder}(R(j,j), R(j+1\colon last, j))$
$\quad$ % Update R
$\quad R(j,j) = R(j,j) - \tau(j)R(j,j) - \tau(j)V(1\colon last-j, j)^T R(j+1\colon last, j)$
$\quad$ if $j < n-p$
$\quad\quad R(j\colon last, j+1\colon n-p) = R(j\colon last, j+1\colon n-p)$
$\quad\quad\quad -\tau(j) \begin{bmatrix} 1 \\ V(1\colon last-j, j) \end{bmatrix}$
$\quad\quad\quad * ([\,1 \quad V(1\colon last-j, j)^T\,]\, R(j\colon last, j+1\colon n-p))$
$\quad$ end
end
$\widetilde{R} =$ upper triangular part of $R(1\colon m, 1\colon n-p)$

If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.18 *Given the matrix V and vector τ from Algorithm 4.2.17 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{m\times m}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with the kth to $(k+p-1)$st columns deleted.*

lim $= \min(m-1, n-p)$
for $j = k\colon lim$
$last = \min(j+p, m)$
$Q(1\colon m, j\colon last) = Q(1\colon m, j\colon last)$
$\quad -\tau(j)\Big(Q(1\colon m, j\colon last) \begin{bmatrix} 1 \\ V(1\colon last-j, j) \end{bmatrix}\Big)\, [\,1 \quad V(1\colon last-j, j)^T\,]$
end
$\widetilde{Q} = Q$

In the case when $k > \min(m-1, n-p)$ then either $k = n-p+1$ or $m < n$ and the deleted columns are in R_{12} where

$$Q^T \widetilde{A} = [\, R_{11} \quad R_{12}\,],$$

with R_{12} full. There is no computation to do in either case.

See Appendix D for Fortran codes `delcols.f` and `delcolsq.f` for updating R and Q respectively.

4.2.5 Adding Columns

Adding One Column

If we wish to *add* a *variable* to our least squares problem, we have the problem of updating $A = QR$ after adding a column, $u \in \mathbb{R}^m$, in the kth position, $1 \leq k \leq n+1$, of $A = QR$, we can then write

$$\widetilde{A} = [\, A(1\colon m, 1\colon k-1) \quad u \quad A(1\colon m, k\colon n)\,]$$

then

$$Q^T\widetilde{A} = [\, R(1\colon m, 1\colon k-1) \quad v \quad R(1\colon m, k\colon n)\,], \tag{4.2.12}$$

where $v = Q^T u$. For example, with $m = 8$, $n = 6$ and $k = 4$ the right-hand side of Equation (4.2.12) looks like:

$$\begin{bmatrix} + & + & + & + & + & + & + \\ 0 & + & + & + & + & + & + \\ 0 & 0 & + & + & + & + & + \\ 0 & 0 & 0 & + & + & + & + \\ 0 & 0 & 0 & \ominus & \oplus & + & + \\ 0 & 0 & 0 & \ominus & 0 & \oplus & + \\ 0 & 0 & 0 & \ominus & 0 & 0 & \oplus \\ 0 & 0 & 0 & \ominus & 0 & 0 & 0 \end{bmatrix},$$

with the nonzero elements to remain represented with a $+$, the elements to be eliminated are $\ominus$ and the zero elements that can be filled in are shown with a $\oplus$.

Thus we can define $m-k$ Givens matrices, $G(i,j) \in \mathbb{R}^{m\times m}$, to eliminate $v(k+1\colon m)$. We then have

$$(G(k,k+1)^T \ldots G(m-1,m)^T Q^T)\widetilde{A} = \widetilde{Q}^T\widetilde{A} = \widetilde{R},$$

where $\widetilde{R} \in \mathbb{R}^{m\times(n+1)}$ is upper trapezoidal and $\widetilde{Q} \in \mathbb{R}^{m\times m}$ is orthogonal. We

then update b by computing

$$G(k,k+1)^T \ldots G(m-1,m)^T Q^T b = \tilde{d}.$$

This gives the following algorithm.

Algorithm 4.2.19 *Given $A = QR \in \mathbb{R}^{m\times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{m\times(n+1)}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with a, $u \in \mathbb{R}^n$, column inserted in the kth position, $1 \leq k \leq n+1$, and $\tilde{d}$ such that $\|\widetilde{A}x - b\|_2 = \|\widetilde{R}x - \tilde{d}\|_2$. The residual, $\|\tilde{d}(n+1\colon m)\|_2$, is also computed.*

$u = Q^T u$
$\tilde{d} = Q^T b$
for $i = m\colon -1\colon k+1$
  $[c(i), s(i)] = \mathbf{givens}(u(i-1), u(i))$
  $u(i-1) = c(i)u(i-1) - s(i)\widetilde{R}(i)$
  % Update R if there is a nonzero row
  if $i \leq n+1$

$$R(i-1\colon i, i-1\colon n) = \begin{bmatrix} c(i) & s(i) \\ -s(i) & c(i) \end{bmatrix}^T R(i-1\colon i, i-1\colon n)$$

  end
  % Update R

$$\tilde{d}(i-1\colon i) = \begin{bmatrix} c(i) & s(i) \\ -s(i) & c(i) \end{bmatrix}^T \tilde{d}(i-1\colon i)$$

end
if $k = 1$
  $\widetilde{R} =$ upper triangular part of $[\,u \quad R\,]$
else if $k = n+1$
  $\widetilde{R} =$ upper triangular part of $[\,R \quad u\,]$
else
  $\widetilde{R} =$ upper triangular part of $[\,R(1\colon m, 1\colon k-1) \quad u \quad R(1\colon m, k\colon n)\,]$
end
% Compute the residual
$resid = \|\tilde{d}(n+1\colon m)\|_2$

If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.20 *Given the vectors c and s from Algorithm 4.2.19 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{m\times m}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with a column inserted in the kth position.*

for $i = m\colon -1\colon k+1$

$$Q(1\colon m, i-1\colon i) = Q(1\colon m, i-1\colon i) \begin{bmatrix} c(j) & s(j) \\ -s(j) & c(j) \end{bmatrix}$$

end

$\widetilde{Q} = Q$

Adding a Block of Columns

If we add p variables to our problem, that is add a block of p columns, $U \in \mathbb{R}^{m \times p}$, in the kth to $(k+p-1)$st positions of A we can write

$$\widetilde{A} = [\, A(1\colon m, 1\colon k-1) \quad U \quad A(1\colon m, k\colon n) \,]$$

then

$$Q^T \widetilde{A} = [\, R(1\colon m, 1\colon k-1) \quad V \quad R(1\colon m, k\colon n) \,],$$

where $V = Q^T U$. For example, with $m = 12$, $n = 6$, $k = 3$ and $p = 3$ the right-hand side of Equation (4.2.12) looks like:

$$\begin{bmatrix}
+ & + & + & + & + & + & + & + & + \\
0 & + & + & + & + & + & + & + & + \\
0 & 0 & + & + & + & + & + & + & + \\
0 & 0 & \ominus & + & + & \oplus & + & + & + \\
0 & 0 & \ominus & \ominus & + & \oplus & \oplus & + & + \\
0 & 0 & \ominus & \ominus & \ominus & \oplus & \oplus & \oplus & + \\
0 & 0 & \ominus & \ominus & \ominus & 0 & \oplus & \oplus & \oplus \\
0 & 0 & \ominus & \ominus & \ominus & 0 & 0 & \oplus & \oplus \\
0 & 0 & \ominus & \ominus & \ominus & 0 & 0 & 0 & 0 \\
0 & 0 & \ominus & \ominus & \ominus & 0 & 0 & 0 & 0 \\
0 & 0 & \ominus & \ominus & \ominus & 0 & 0 & 0 & 0 \\
0 & 0 & \ominus & \ominus & \ominus & 0 & 0 & 0 & 0
\end{bmatrix},$$

with the nonzero elements to remain represented with a $+$, the elements to be eliminated are $\ominus$ and the zero elements that can be filled in are shown with a $\oplus$.

We would like an orthogonal matrix, W, such that

$$\begin{bmatrix} I & 0 \\ 0 & W^T \end{bmatrix} Q^T \widetilde{A} = \widetilde{R}, \quad W \in \mathbb{R}^{(m-k+1)\times(m-k+1)}.$$

If W were the product of Householder matrices, then $\widetilde{R}$ would be full. Thus we use Givens matrices and generalize Algorithm 4.2.19.

Algorithm 4.2.21 *Given $A = QR \in \mathbb{R}^{m\times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{m\times(n+p)}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with a block of columns, $U \in \mathbb{R}^{m\times p}$, inserted in the kth to $(k+p-1)$st position, $1 \leq k \leq n+1$, $p \geq 1$, and $\tilde{d}$ such that $\|\widetilde{A}x - b\|_2 = \|\widetilde{R}x - \tilde{d}\|_2$. The residual, $\|\tilde{d}(n+1\colon m)\|_2$, is also computed.*

$U = Q^T U$
$\tilde{d} = Q^T b$
for $j = 1\colon p$
  for $i = m\colon -1\colon k+j$
    $[C(i,j), S(i,j)] = \mathbf{givens}(U(i-1,j), U(i,j))$
    % Update U
    $U(i-1,j) = C(i,j)U(i-1,j) - S(i,j)U(i,j)$
    if $j < p$
      $U(i-1\colon i, j+1\colon p) =$
        $$\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T U(i-1\colon i, j+1\colon p)$$
    end
    % Update R if there is a nonzero row
    if $i \leq n+j$
      $R(i-1\colon i, i-j\colon n) =$
        $$\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T R(i-1\colon i, i-j\colon n)$$
    end
    % Update $\tilde{d}$
    $$\tilde{d}(i-1\colon i) = \begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T \tilde{d}(i-1\colon i)$$
  end
end
if $k = 1$
  $\widetilde{R} =$ upper triangular part of $[U \quad R]$
else if $k = n+1$

$\widetilde{R} =$ upper triangular part of $[\, R \quad U \,]$
else
$\widetilde{R} =$ upper triangular part of $[\, R(1\colon m, 1\colon k-1) \quad U \quad R(1\colon m, k\colon n) \,]$
end
% Compute the residual
$resid = \|\tilde{d}(n+1\colon m)\|_2$

Computing $\widetilde{R}$ requires $6(mp(n+p-m/2)-p^2(n/2-k/2-p/3)+kp(k/2-n))$ flops, versus $2(n+p)^2(m-(n+p)/3)$ for the Householder QR factorization of $\widetilde{A}$. If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.22 *Given matrices C and S from Algorithm 4.2.21 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{m\times m}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with a block of columns inserted in the kth to $(k+p-1)$st positions.*

for $j = 1\colon p$
for $i = m\colon -1\colon k+j$

$$Q(1\colon m, i-1\colon i) = Q(1\colon m, i-1\colon i) \begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}$$

end
end
$\widetilde{Q} = Q$

We can improve on this algorithm by including a Level 3 BLAS part by using a blocked QR factorization of part of $\widetilde{A}$ before we finish the elimination

process with Givens matrices. That is, for our example:

$$\begin{bmatrix}
+ & + & + & + & + & + & + & + & + \\
0 & + & + & + & + & + & + & + & + \\
0 & 0 & + & + & + & + & + & + & + \\
0 & 0 & \ominus & + & + & \oplus & + & + & + \\
0 & 0 & \ominus & \ominus & + & \oplus & \oplus & + & + \\
0 & 0 & \ominus & \ominus & \ominus & \oplus & \oplus & \oplus & + \\
0 & 0 & \ominus & \ominus & \ominus & 0 & \oplus & \oplus & \oplus \\
0 & 0 & \odot & \ominus & \ominus & 0 & 0 & \oplus & \oplus \\
0 & 0 & \odot & \odot & \ominus & 0 & 0 & 0 & \oplus \\
0 & 0 & \odot & \odot & \odot & 0 & 0 & 0 & 0 \\
0 & 0 & \odot & \odot & \odot & 0 & 0 & 0 & 0 \\
0 & 0 & \odot & \odot & \odot & 0 & 0 & 0 & 0
\end{bmatrix},$$

we eliminate the elements shown with a $\odot$ with a QR factorization of the bottom 6 by 3 block of V and the remainder of the elements can be eliminated with Givens matrices and are shown with a $\ominus$. The zero elements that can be filled in are shown with a $\oplus$ and the nonzero elements to remain represented with a $+$ as before.

For the case of $k \neq 1, n+1$ and $m > n+1$, we have

$$Q^T \widetilde{A} = \begin{bmatrix} R_{11} & V_{12} & R_{12} \\ 0 & V_{22} & R_{23} \\ 0 & V_{32} & 0 \end{bmatrix}$$

where $R_{11} \in \mathbb{R}^{(k-1)\times(k-1)}$ and $R_{23} \in \mathbb{R}^{(n-k+1)\times(n-k+1)}$ are upper triangular, then if V_{32} has the QR factorization $V_{32} = Q_V R_V \in \mathbb{R}^{(m-n)\times p}$ we have

$$\begin{bmatrix} I_n & 0 \\ 0 & Q_V^T \end{bmatrix} Q^T \widetilde{A} = \begin{bmatrix} R_{11} & V_{12} & R_{12} \\ 0 & V_{22} & R_{23} \\ 0 & R_V & 0 \end{bmatrix}.$$

We then eliminate the upper triangular part of R_V and the lower triangular part of V_{22} with Givens matrices which makes R_{23} full and the bottom right

block upper trapezoidal. So we have finally

$$G(k+2p-2,k+2p-1)^T \quad \ldots \quad G(k+p,k+p+1)^T G(k,k+1)^T$$
$$\ldots \quad G(k+p-1,k+p)^T \begin{bmatrix} I_n & 0 \\ 0 & Q_V^T \end{bmatrix} Q^T \widetilde{A} = \widetilde{R}$$

This gives the following algorithm.

Algorithm 4.2.23 *Given $A = QR \in \mathbb{R}^{m\times n}$, with $m \geq n$, this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{m\times(n+p)}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with a block of columns, $U \in \mathbb{R}^{m\times p}$, inserted in the kth to $(k+p-1)$st position, $1 \leq k \leq n+1$, $p \geq 1$, and $\tilde{d}$ such that $\|\widetilde{A}x - b\|_2 = \|\widetilde{R}x - \tilde{d}\|_2$. The residual, $\|\tilde{d}(n+1\colon m)\|_2$, is also computed. The algorithm incorporates a Level 3 QR factorization.*

$U = Q^T U$
$\tilde{d} = Q^T b$
if $m > n+1$
    % Factorize rows $n+1$ to m of U if there are more than 1,
    % with a Level 3 QR algorithm
    $U(n+1\colon m, 1\colon p) = Q_U R_U$
    % Update $\tilde{d}$
    $\tilde{d}(n+1\colon m) = Q_U^T \tilde{d}(n+1\colon m)$
end
if $k \leq n$
    % Zero out the rest with Givens
    for $j = 1\colon p$
        % First iteration updates one column
        $upfirst = n$
        for $i = n+j\colon -1\colon j+1$
            $[C(i,j), S(i,j)] = \mathbf{givens}(U(i-1,j), U(i,j))$
            % Update U
            $U(i-1,j) = C(i,j)U(i-1,j) - S(i,j)U(i,j)$
            if $j < p$
                $U(i-1\colon i, j+1\colon p) =$
                $$\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T U(i-1\colon i, j+1\colon p)$$
            end
            % Update R

$R(i-1\colon i, upfirst\colon n) =$

$$\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T R(i-1\colon i, upfirst\colon n)$$

% Update one more column next i step

$upfirst = upfirst - 1$

% Update $\tilde{d}$

$\tilde{d}(i-1\colon i) = \begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T \tilde{d}(i-1\colon i)$

end

end

end

if $k = 1$

$\widetilde{R} =$ upper triangular part of $[\,U \quad R\,]$

else if $k = n+1$

$\widetilde{R} =$ upper triangular part of $[\,R \quad U\,]$

else

$\widetilde{R} =$ upper triangular part of $[\,R(1\colon m, 1\colon k-1) \quad U \quad R(1\colon m, k\colon n)\,]$

end

% Compute the residual

$resid = \|\tilde{d}(n+1\colon m)\|_2$

If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.24 *Given matrices Q_U, C and S and the vector τ from Algorithm 4.2.23 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{m\times m}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with a block of columns inserted in the kth to $(k+p-1)$st positions.*

if $m > n+1$

$Q(1\colon m, 1\colon m-n) = Q(1\colon m, 1\colon m-n)Q_U$

end

if $k \le n$

for $j = 1\colon p$

for $i = n+j\colon -1\colon j+1$

$$Q(1\colon m, i-1\colon i) = Q(1\colon m, i-1\colon i)\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}$$

end

end

end

$\widetilde{Q} = Q$

Updating the QR Factorization for any m and n

In the case where $m > n$, to update only the QR factorization, then we need to consider the limits of the for loops and *upstart* to convert Algorithm 4.2.23.

- We introduce $jstop$ which is the last index in the outer for loop. There may be a situation where there are not elements to eliminate over the full width of U. For example $Q^T\widetilde{A}$, for $m = 5$, $n = 6$, $k = 3$ and $p = 3$, looks like:

$$\begin{bmatrix} + & + & + & + & + & + & + & + & + \\ 0 & + & + & + & + & + & + & + & + \\ 0 & 0 & + & + & + & + & + & + & + \\ 0 & 0 & \ominus & + & + & \oplus & + & + & + \\ 0 & 0 & \ominus & \ominus & + & \oplus & \oplus & + & + \end{bmatrix},$$

  and there are no elements to eliminate in the last column of V.

- $istart$ is introduced as the first element in the jth column to be eliminated cannot exceed m.

- The first column to be updated for jth step may no longer be n, so $upfirst$ is set accordingly.

Note if $m \leq n + 1$ and $k > n$ there is nothing to do and neither outer if block is entered.

Algorithm 4.2.25 *Given $A = QR \in \mathbb{R}^{m\times n}$, this algorithm computes $\widetilde{Q}^T\widetilde{A} = \widetilde{R} \in \mathbb{R}^{m\times(n+p)}$ where $\widetilde{R}$ is upper trapezoidal, $\widetilde{Q}$ is orthogonal and $\widetilde{A}$ is A with a block of columns, $U \in \mathbb{R}^{m\times p}$, inserted in the kth to $(k + p - 1)$st position. The algorithm incorporates a Level 3 QR factorization.*

$U = Q^T U$
if $m > n + 1$
    % Factorize rows $n + 1$ to m of U if there are more than 1,
    % with a Level 3 QR algorithm
    $U(n + 1\colon m, 1\colon p) = Q_U R_U$

end
if $k \leq n$
    % Zero out the rest with Givens, stop at the last column of
    % U or the last row if that is reached first
    $jstop = \min(p, m-k-2)$
    for $j = 1{:}\, jstop$
        % Start at first row to be eliminated in current column
        $istart = \min(n+j, m)$
        % Index of first nonzero column in update of R
        $upfirst = \max(istart - j - 1, 1)$
        for $i = istart{:}\, -1{:}\, j+1$
            $[C(i,j), S(i,j)] = \mathbf{givens}(U(i-1,j), U(i,j))$
            % Update U
            $U(i-1,j) = C(i,j)U(i-1,j) - S(i,j)U(i,j)$
            if $j < p$
                % Update U
                $U(i-1{:}\, i, j+1{:}\, p) =$
                $$\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T U(i-1{:}\, i, j+1{:}\, p)$$
            end
            % Update R
                $R(i-1{:}\, i, upfirst{:}\, n) =$
                $$\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}^T R(i-1{:}\, i, upfirst{:}\, n)$$
            % Update one more column next i step
            $upfirst = upfirst - 1$
        end
    end
end
if $k = 1$
    $\widetilde{R} =$ upper triangular part of $[\, U \quad R \,]$
else if $k = n+1$
    $\widetilde{R} =$ upper triangular part of $[\, R \quad U \,]$
else
    $\widetilde{R} =$ upper triangular part of $[\, R(1{:}\, m, 1{:}\, k-1) \quad U \quad R(1{:}\, m, k{:}\, n) \,]$
end

If $\widetilde{Q}$ is required, it can be computed with the following algorithm.

Algorithm 4.2.26 *Given matrices Q_U, C and S and the vector τ from Algorithm 4.2.25 this algorithm forms an orthogonal matrix $\widetilde{Q} \in \mathbb{R}^{m\times m}$ such that $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is the matrix $A = QR$ with a block of columns inserted in the kth to $(k+p-1)$st positions.*

if $m > n+1$
  $Q(1\colon m, n+1\colon m) = Q(1\colon m, n+1\colon m)Q_U$
end
if $k \le n$
  $jstop = \min(p, m-k-2)$
  for $j = 1\colon jstop$
    $istart = \min(n+j, m)$
    for $i = istart\colon -1\colon j+1$

$$Q(1\colon m, i-1\colon i) = Q(1\colon m, i-1\colon i)\begin{bmatrix} C(i,j) & S(i,j) \\ -S(i,j) & C(i,j) \end{bmatrix}$$

    end
  end
end
$\widetilde{Q} = Q$

See Appendix D for Fortran codes `addcols.f` and `addcolsq.f` for updating R and Q respectively.

4.3 Error Analysis

It is well known that orthogonal transformations are stable. We have the following columnwise results [29], where

$$\tilde{\gamma}_k = \frac{cku}{1-cku},$$

and u is the unit roundoff and c is a small integer constant.

Lemma 4.3.1 (Sequence of Givens Matrices) *If*

$$B = G_r \ldots G_1 A = Q^T A \in \mathbb{R}^{n\times n}$$

where G_i is a Givens matrix, then the computed matrix $\widehat{B}$ satisfies

$$Q^T A - \widehat{B} = \Delta B, \qquad \|\Delta b_j\|_2 \leq \tilde{\gamma}_r \|a_j\|_2, \quad j = 1\colon n. \quad \square \tag{4.3.1}$$

Lemma 4.3.2 (Sequence of Householder Matrices) *If*

$$B = H_r \ldots H_1 A = Q^T A \in \mathbb{R}^{n \times n}$$

where H_i is a Householder matrix, then the computed matrix $\widehat{B}$ satisfies

$$Q^T A - \widehat{B} = \Delta B, \qquad \|\Delta b_j\|_2 \leq \tilde{\gamma}_{nr} \|a_j\|_2, \quad j = 1\colon n. \quad \square \tag{4.3.2}$$

This result implies that Householder transformations are less accurate by a factor of n, but this is not observed in practice. We then have

Theorem 4.3.1 (Householder QR Factorization) *If*

$$R = Q^T A$$

where Q is a product of Householder matrices, then the computed factor $\widehat{R}$ satisfies

$$Q^T A - \widehat{R} = \Delta R, \qquad \|\Delta r_j\|_2 \leq \tilde{\gamma}_{mn} \|a_j\|_2, \quad j = 1\colon n. \quad \square$$

We now give results for computing the factor $\widetilde{R}$ by our algorithms.

4.3.1 Deleting Rows

We have from Section 4.2.1

$$G(1,2)^T \ldots G(m-1,m)^T \widehat{R} = \begin{bmatrix} v^T \\ \widetilde{R} \end{bmatrix},$$

and from (4.3.1) we have for the computed quantities $\widehat{\widetilde{R}}$ and $\hat{v}$

$$\widehat{\widetilde{R}} = \widetilde{R} + \Delta R, \qquad \|\Delta r_j\|_2 \leq \tilde{\gamma}_{mp} \left\| \begin{bmatrix} v^T(j) \\ \hat{r}(1\colon n, j) \end{bmatrix} \right\|_2, \quad j = 1\colon n.$$

Recall that the $G(i,j)$ are chosen to introduce zeros in Q.

4.3.2 Adding Rows

We have from Section 4.2.3

$$H_n \dots H_1 \begin{bmatrix} \widehat{R} \\ U \end{bmatrix} = \widetilde{R}, \quad U \in \mathbb{R}^{p \times n},$$

and from (4.3.2) we have

$$\widehat{\widetilde{R}} = \widetilde{R} + \Delta R, \qquad \|\Delta r_j\|_2 \le \tilde{\gamma}_{n(p+1)} \left\| \begin{bmatrix} \hat{r}_{jj} \\ U(:, j) \end{bmatrix} \right\|_2, \quad j = 1\colon n.$$

4.3.3 Deleting Columns

We have from Section 4.2.4

$$H_{n-p} \dots H_k \, [\, \widehat{R}(:, 1\colon k-1) \quad \widehat{R}(:, k+p\colon n) \,] = \widetilde{R},$$

and from (4.3.2) we have

$$\widehat{\widetilde{R}} = \widetilde{R} + \begin{bmatrix} 0 \\ \Delta R \end{bmatrix}, \quad \Delta R \in \mathbb{R}^{(m-k+1) \times n}$$

$$\begin{aligned} \|\Delta r_j\|_2 &= 0, && j = 1\colon k-1, \\ &\le \tilde{\gamma}_{(n-k-p+1)(n-k+1)} \|\hat{r}(k\colon n, j)\|_2, && j = k\colon n-p. \end{aligned}$$

4.3.4 Adding Columns

We have from Section 4.2.5

$$\begin{aligned} &G(k+2p-2, k+2p-1) \dots \\ &G(k+p-1, k+p) \begin{bmatrix} I & 0 \\ 0 & Q_V^T \end{bmatrix} [\, \widehat{R}(:, 1\colon k-1) \quad \widehat{V} \quad \widehat{R}(:, k\colon n) \,] = \widetilde{R}, \end{aligned}$$

where $\widehat{V} = Q^T U \in \mathbb{R}^{m \times p}$ and from (4.3.1) and (4.3.2) we have

$$\widehat{\widetilde{R}} = \widetilde{R} + \begin{bmatrix} 0 \\ 0 \\ \Delta H \end{bmatrix} + \begin{bmatrix} 0 \\ \\ \Delta G \end{bmatrix}, \quad \Delta H \in \mathbb{R}^{(m-n) \times n}, \quad \Delta G \in \mathbb{R}^{(m-k+1) \times n}$$

$$\begin{aligned}
\|\Delta H_j\|_2 &= 0, && k-1 \geq j \geq k+p, \\
&\leq \tilde{\gamma}_{(n-k)p}\|\widehat{V}(n+1\colon m, j)\|_2, && j = k\colon k+p-1,
\end{aligned}$$

$$\begin{aligned}
\|\Delta G_j\|_2 &= 0, && j = 1\colon k-1, \\
&\leq \tilde{\gamma}_{(n-k)n}\left\|(Q^T\widehat{V})(k\colon m, j) + \begin{bmatrix} 0 \\ \Delta\hat{r}_j \end{bmatrix}\right\|_2, && j = k\colon n+p.
\end{aligned}$$

Given these results we expect the normwise backward error

$$\frac{\|\widetilde{A} - \widetilde{Q}\widetilde{R}\|_2}{\|\widetilde{A}\|_2},$$

when $\widetilde{Q}$ and $\widetilde{R}$ are computed with our algorithms to be close to that with $\widetilde{Q}$ and $\widetilde{R}$ computed directly from $\widetilde{A}$. We consider some examples in the next section.

4.4 Numerical Experiments

4.4.1 Speed Tests

In this section we test the speed of our double precision Fortran 77 codes, see Appendix D, against LAPACK's `DGEQRF`, a Level 3 BLAS routine for computing the QR factorization of a matrix. The input matrix, in this case $\widetilde{A}$, is overwritten with $\widetilde{R}$, and $\widetilde{Q}$ is returned in factored form in the same way as our codes do.

The tests were performed on a 1400MHz AMD Athlon running Red Hat Linux version 6.2 with kernel 2.2.22. The unit roundoff $u \approx$ 1.1e-16.

We tested our code with

$$m = \{1000,\ 2000,\ 3000,\ 4000,\ 5000\}$$

and $n = 0.3m$, and the number of columns added or deleted was $p = 100$. We generated our test matrices by populating an array with random double

precision numbers generated with the LAPACK auxiliary routine `DLARAN`. $A = QR$ was computed with `DGEQRF`, and $\widetilde{A}$ was formed appropriately.

We timed our codes acting on $Q^T\widetilde{A}$, the starting point for computing $\widetilde{R}$, and in the case of adding columns we included the computation of Q^TU in our timings, which we formed with the BLAS routine `DGEMM`. We also timed `DGEQRF` acting on only the part of $Q^T\widetilde{A}$ that needs to be updated, the nonzero part from row and column k onwards. Here we can construct $\widetilde{R}$ with this computation and the original R. Finally, we compute `DGEQRF` acting on $\widetilde{A}$. We aim to show our codes are faster than these alternatives. In all cases an average of three timings are given.

To test our code `DELCOLS` we first chose $k = 1$, the position of the first column deleted, where the maximum amount of work is required to update the factorization. We have

$$\widetilde{A} = A(1\colon m, p+1\colon n), \quad \text{and} \quad Q^T\widetilde{A} = R(1\colon m, p+1\colon n)$$

and timed:

- `DGEQRF` on $\widetilde{A}$.
- `DGEQRF` on $(Q^T\widetilde{A})(k\colon n, k\colon n-p)$ which computes the nonzero entries of $\widetilde{R}(k\colon m, p+1\colon n)$.
- `DELCOLS` on $Q^T\widetilde{A}$.

The results are given in Figure 4.4.1. Our code is clearly much faster than recomputing the factorization from scratch with `DGEQRF`, and for $n = 5000$ there is a speedup of 20. Our code is also faster than using `DGEQRF` on $(Q^T\widetilde{A})(k\colon n, k\colon n-p)$, where there is a maximum speedup of over 3.

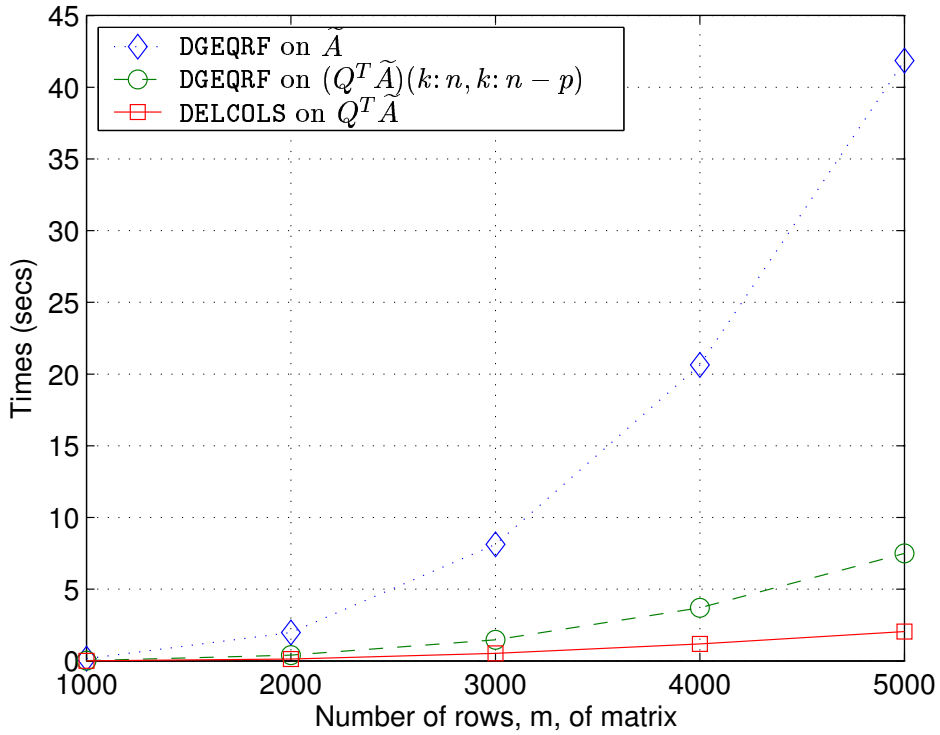

Figure 4.4.1: Comparison of speed for `DELCOLS` with $k = 1$ for different m.

We then tested for $k = n/2$ where much less work is required to perform the updating, we have

$$\begin{aligned} \widetilde{A} &= [\, A(1{:}\,m, 1{:}\,k-1) \quad A(1{:}\,m, k+p{:}\,n)\,], \quad \text{and} \\ Q^T\widetilde{A} &= [\, R(1{:}\,m, 1{:}\,k-1) \quad R(1{:}\,m, k+p{:}\,n)\,] \end{aligned}$$

and timed:

- `DGEQRF` on $(Q^T\widetilde{A})(k{:}\,n, k{:}\,n-p)$ which computes the nonzero entries of $\widetilde{R}(k{:}\,m, k{:}\,n-p)$.
- `DELCOLS` on $Q^T\widetilde{A}$.

The results are given in Figure 4.4.2. The timings for `DGEQFR` on $\widetilde{A}$ would, of course, be the same as for $k = 1$, giving a maximum speedup of over 100

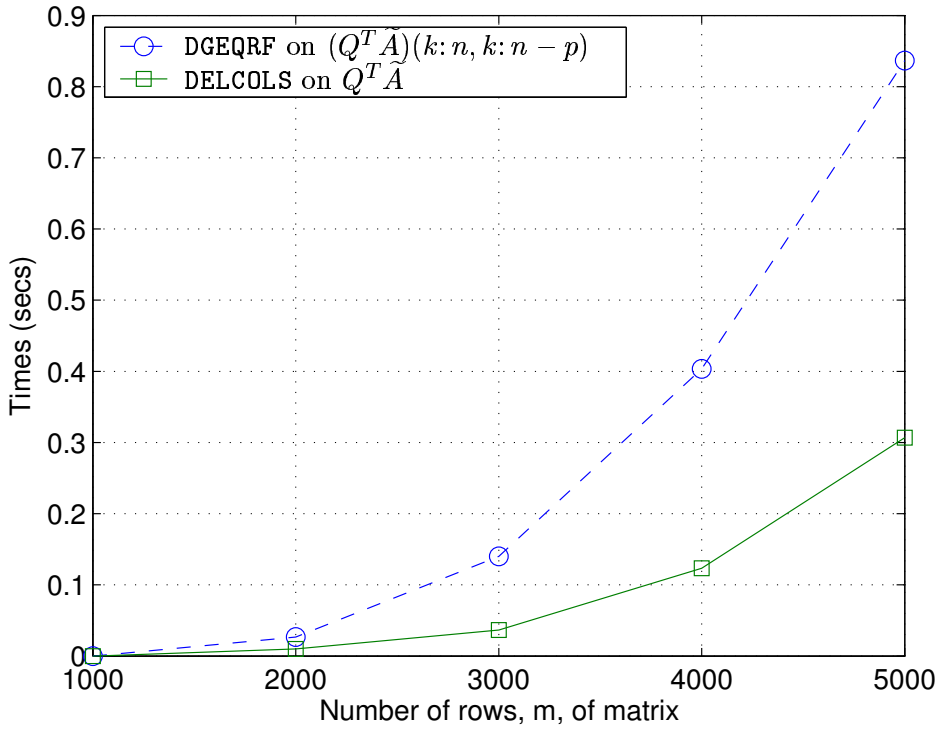

Figure 4.4.2: Comparison of speed for `DELCOLS` with $k = n/2$ for different m.

in this case. We achieve a speedup of approximately 3 over using `DGEQRF` on $(Q^T\widetilde{A})(k{:}\,n, k{:}\,n-p)$.

We then considered the effect of varying p with `DELCOLS` for fixed $m = 3000$, $n = 1000$ and $k = 1$. As we delete more columns from A there are less columns to update, but more work is required for each one. We chose

$$p = \{100,\ 200,\ 300,\ 400,\ 500,\ 600\ 700,\ 800\}$$

and timed:

- `DGEQRF` on $\widetilde{A}$
- `DGEQRF` on $(Q^T\widetilde{A})(k{:}\,n, k{:}\,n-p)$ which computes the nonzero entries of $\widetilde{R}(k{:}\,m, k{:}\,n-p)$.

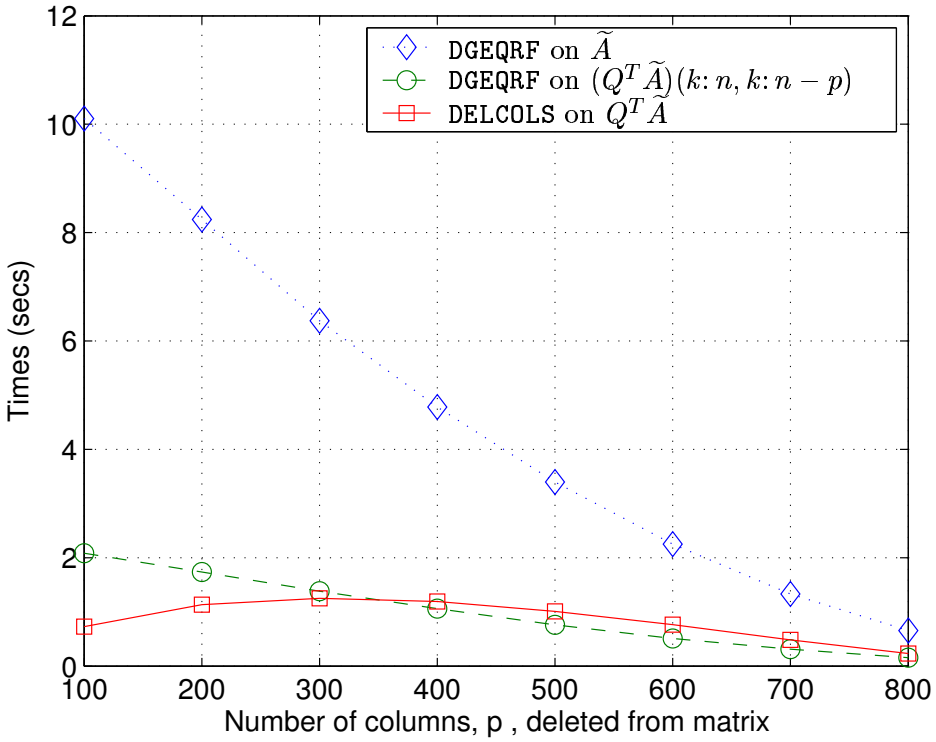

Figure 4.4.3: Comparison of speed for DELCOLS for different p.

- DELCOLS on $Q^T\widetilde{A}$.

The results are given in Figure 4.4.3. The timings for DELCOLS are relatively level and peak at $p = 300$, whereas the timings for the other codes obviously decrease with p. The speedup of our code decreases with p, and from $p = 300$ there is little difference between our code and DGEQRF on $(Q^T\widetilde{A})(k\colon n, k\colon n-p)$.

To test ADDCOLS we generated random matrices $A \in \mathbb{R}^{m\times n}$ and $U \in \mathbb{R}^{m\times p}$, and again use

$$m = \{1000,\ 2000,\ 3000,\ 4000,\ 5000\}$$

$n = 0.3m$, and $p = 100$. We first set $k = 1$ where maximum updating is required. We have

$$\widetilde{A} = [\,U \quad A\,], \quad \text{and} \quad Q^T\widetilde{A} = [\,Q^TU \quad R\,]$$

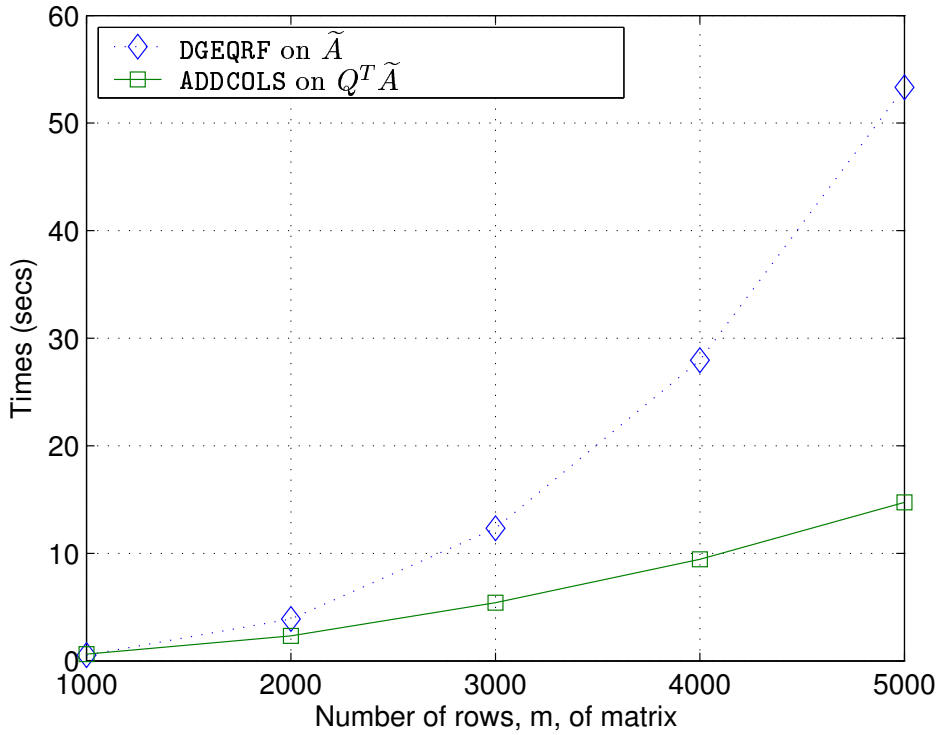

Figure 4.4.4: Comparison of speed for ADDCOLS with $k = 1$ for different m.

and timed:

- DGEQRF on $\widetilde{A}$.
- ADDCOLS on $Q^T\widetilde{A}$, including the computation of Q^TU with DGEMM.

The results are given in Figure 4.4.4. Here our code achieves a speedup of over 3 for $m = 5000$ over the complete factorization of $\widetilde{A}$.

We then tested for $k = n/2$, where less work is required to do the updating. We have

$$\begin{aligned} \widetilde{A} &= [\, A(1\!:\!m, 1\!:\!k-1) \quad U \quad A(1\!:\!m, k\!:\!n)\,], \quad \text{and} \\ Q^T\widetilde{A} &= [\, R(1\!:\!m, 1\!:\!k-1) \quad Q^TU \quad R(1\!:\!m, k\!:\!n)\,] \end{aligned}$$

and timed:

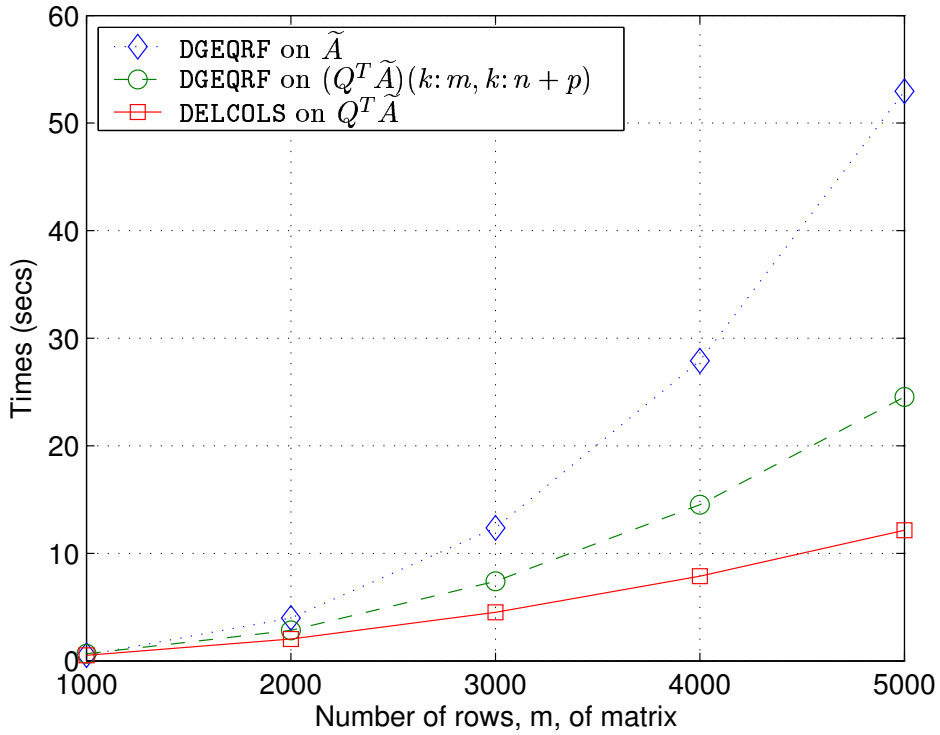

Figure 4.4.5: Comparison of speed for `ADDCOLS` with $k = n/2$ for different m.

- `DGEQRF` on $\widetilde{A}$, as above.
- `DGEQRF` on $(Q^T\widetilde{A})(k\colon m, k\colon n+p)$ which computes $\widetilde{R}(k\colon m, k\colon n+p)$, including the computation of Q^TU for which we again use `DGEMM`.
- `ADDCOLS` on $Q^T\widetilde{A}$, including the computation of Q^TU.

The results are given in Figure 4.4.5. Here we have a maximum speedup of over 4 with our code against `DGEQRF` on $\widetilde{A}$. We achieve a maximum speedup of approximately 2 against `DGEQRF` on $(Q^T\widetilde{A})(k\colon m, k\colon n+p)$.

We do not vary p as this increases the work for both our code and `DGEQRF` on $(Q^T\widetilde{A})(k\colon m, k\colon n+p)$ roughly equally.

4.4.2 Backward Error Tests

The tests here were performed on a 2545MHz AMD Pentium running a hybrid version of Red Hat Linux 8 and 9 with kernel 2.4.20.

Here we test our code for updating Q and R; `DELCOLS` and `DELCOLSQ` for deleting columns and `ADDCOLS` and `ADDCOLSQ` for adding columns. We did this in the following way:

- We form a random matrix

$$A^{(0)} = [\, A_1 \quad U \quad A_2 \,], \quad \|A_1\|_F, \|A_2\|_F, \|U\|_F \text{ of order } 100,$$

  where $A_1 \in \mathbb{R}^{m\times(k-1)}$, $A_2 \in \mathbb{R}^{m\times(n-k-p+1)}$, $U \in \mathbb{R}^{m\times p}$.

- We then form the QR factorization

$$A^{(0)} = Q^{(0)}R^{(0)} = Q^{(0)}\,[\, R_1 \quad R_U \quad R_2 \,],$$

  where $R_1 \in \mathbb{R}^{m\times(k-1)}$, $R_2 \in \mathbb{R}^{m\times(n-k-p+1)}$, $R_U \in \mathbb{R}^{m\times p}$, using the LAPACK routines `DGEQRF` and `DORGQR`.

- Next, for

$$\widetilde{A} = [\, A_1 \quad A_2 \,],$$

  we form

$$Q^{(0)^T}\widetilde{A} = [\, R_1 \quad R_2 \,],$$

  and call `DELCOLS` and `DELCOLSQ` to update the QR factorization of $\widetilde{A}$, forming

$$\widetilde{A} = \widetilde{Q}\widetilde{R} = [\, \widetilde{R}_1 \quad \widetilde{R}_2 \,],$$

  where $\widetilde{R}_1 \in \mathbb{R}^{m\times(k-1)}$, $\widetilde{R}_2 \in \mathbb{R}^{m\times(n-k-p+1)}$.

- We now compute the QR factorization of $A^{(0)}$ by updating $\widetilde{Q}$ and $\widetilde{R}$. We call `ADDCOLS` on

$$[\,\widetilde{R}_1 \quad \widetilde{Q}^T U \quad \widetilde{R}_2\,]$$

to form $R^{(1)}$ and then call `ADDCOLSQ` to form $Q^{(1)}$, so we have, in exact arithmetic

$$A^{(0)} = Q^{(1)} R^{(1)}.$$

We then repeat this *rep* times and measure the normwise backward error

$$\frac{\|A^{(0)} - Q^{(rep)} R^{(rep)}\|_2}{\|A\|_2}.$$

We use every combination of the following set of parameters:

$$\begin{aligned} m &= 500 \\ n &= \{400,\ 500,\ 600\} \\ p &= \{50,\ 100,\ 150\} \\ k &= \{1,\ 51, \ldots,\ n - p + 1\}. \end{aligned}$$

We then repeated the entire process, but with

$$\|U\|_F \text{ of order 1e+9.}$$

The results are given in Table 4.4.1 and Table 4.4.2. The error increases with the number of repeats which is expected. However, the value is not effected significantly by the value of $\|U\|_F$.

The smallest value of the error in every case was approximately of order $10u$. The worse case was still only of order $2 * rep * u$.

Table 4.4.1: Normwise backward error for $\|U\|_F$ order 100.

rep	5	50	500
Smallest error over all tests	1.146e-15	1.212e-15	1.223e-15
Largest error over all tests	5.031e-15	2.399e-14	1.252e-13

Table 4.4.2: Normwise backward error for $\|U\|_F$ order 1e+9.

rep	5	50	500
Smallest error over all tests	8.298e-16	9.309e-16	9.576e-16
Largest error over all tests	4.381e-15	2.055e-14	1.014e-13

4.5 Conclusions

The speed tests show that our updating algorithms are faster than computing the QR factorization from scratch or using the factorization to update columns k onward, the only columns needing updating.

Furthermore, the normwise backward error tests show that the errors are within the bound for computing the Householder QR factorization of $\widetilde{A}$. Thus, within the parameters of our experiments, the increase of speed is not at the detriment of accuracy.

We propose the double precision Fortran 77 codes `delcols.f`, `delcolsq.f`, `addcols.f` and `addcolsq.f`, and their single precision and complex equivalents, be included in LAPACK.

4.6 Software Available

Here we list some software that is available to update the QR factorization and least squares problem. An '`x`' in a routine indicates more than one routine for different precisions or for real or complex data.

4.6.1 LINPACK

LINPACK [21] has three routines that update the least squares problem and the QR factorization.

- `xCHUD` updates the least squares problem when a row has been added in the $(m+1)$st position.
- `xCHDD` updates the least squares problem when a row has been deleted from the mth position, an implementation of Saunder's algorithm.
- `xCHEX` update the least squares problem when the rows of A have been permuted.

  In all cases the transformation matrices are represented by a vectors of sines and cosines, and $\widetilde{Q}$ is not constructed.

4.6.2 MATLAB

MATLAB [36] supply three routines for updating the QR factorization only.

- `qrdelete` updates when one row or column is deleted from any position.
- `qrinsert` updates when one row or column is added to any position.
- `qrupdate` returns the factorization of A after a rank one change, that is

$$\widetilde{A} = A + uv^T, \quad u \in \mathbb{R}^m,\ v \in \mathbb{R}^n.$$

  In all cases both $\widetilde{Q}$ and $\widetilde{R}$ are returned.

4.6.3 The NAG Library

The Mark 20 NAG Library [38] contains routines for updating two cases.

- `F06xPF` performs the factorization

$$\alpha u v^T + R_1 = \overline{Q}\widetilde{R}_1,$$

where

$$R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}, \quad \widetilde{R} = \begin{bmatrix} \widetilde{R}_1 \\ 0 \end{bmatrix}, \quad R_1, \widetilde{R}_1 \in \mathbb{R}^{n \times n},$$

and

$$\widetilde{Q} = Q\overline{Q}.$$

$\overline{Q}$ is represented by vectors of sines and cosines.

- `F06xQF` performs the downdating problem

$$\begin{bmatrix} R_1 \\ \alpha v^T \end{bmatrix} = \overline{Q} \begin{bmatrix} \widetilde{R}_1 \\ 0 \end{bmatrix},$$

where R, $\widetilde{R}$ and $\widetilde{Q}$ are as above.

4.6.4 Reichel and Gragg's Algorithms

Reichel and Gragg [43] provide several Fortran 77 implementations of the algorithms discussed in [13] for updating the QR factorization, returning both $\widetilde{Q}$ and $\widetilde{R}$. In all cases only $m \geq n$ is handled. The routines use BLAS like routines for matrix and vector operations written for optimal performance on the test machine used in [43]. No error results are given for the Fortran routines, although some results are given for the Algol implementations in [13].

- `DDELR` updates after one row is deleted; this algorithm varies from ours and uses a Gram-Schmidt re-orthogonalization process.

- `DINSR` updates when one row is added, and is similar to our algorithm.

- `DDELC` updates when one column is deleted, and is similar to our algorithm.

- `DINSC` updates after one column is added; this algorithm varies from ours and again uses a Gram-Schmidt re-orthogonalization process.
- `DRNK1` updates after a rank 1 modification to A.
- `DRRPM` updates when $\widetilde{A}$ is A with some if its columns permuted.

4.6.5 What's new in our algorithms

Our contribution is:

- We deal with adding/deleting block of p row/columns, and in two of the four cases we exploit the Level 3 BLAS. Also, the Level 2 code for deleting a block of rows is more efficient than calling the code for deleting one row p times.
- In the case of updating the QR factorization we place no restrictions on m and n.
- All our codes call existing BLAS and LAPACK routines.

Chapter 5

Summary

We have presented new algorithms and Fortran 77 LAPACK-style codes for computing the Cholesky factorization with complete pivoting of a symmetric positive semidefinite matrix, namely:

- `lev2pchol.f` A Level 2 BLAS routine.
- `lev3pchol.f` A Level 3 BLAS routine.

It has been shown that these new codes can be many times faster than the existing LINPACK code. Also, with the Higham [29] stopping criterion they provide more reliable rank detection and can have a smaller normwise backward error than using the existing LINPACK code.

We propose our codes should be included in a future release of LAPACK, and have written testing code for this purpose.

The semidefinite symmetric generalized eigenvalue problem has been considered, both the regular and nonregular cases. We presented an algorithm that can have a potentially smaller operation count than existing methods. Different rank revealing factorizations were explored for transforming the problem and demonstrated practically with our MATLAB M-file `ssgep.m`. In particular we

did not restrict ourselves to orthogonal factorizations as other methods have done.

Our numerical experiments showed there is a trade-off between the number of flops and the size of the backward error of the solution. That is, the implementation of our algorithm with orthogonal rank revealing factorizations required more flops than using nonorthogonal rank revealing factorizations but the normwise backward error was generally smaller.

For regular matrix pencils the normwise backward error was generally better for the QZ algorithm than the implementations of our algorithm, but required much more flops. As the QZ algorithm can not be relied upon for nonregular pencils, we compared our algorithm with the GUPTRI algorithm in this case. However, we found the GUPTRI algorithm sometimes failed and always gave a larger normwise backward error. It also needs many times more flops than the implementations of our algorithm. Thus, for nonregular matrix pencil our algorithm performs better in every regard for out test problems.

There is still the open question as to the effect of the rank decisions we make at each step of our algorithm and are made in the GUPTRI algorithm on the final solution of the semidefinite symmetric generalized eigenvalue problem.

Updating the QR factorization with applications to the least squares problem was also treated. Algorithms were presented that compute the factorization $\widetilde{A} = \widetilde{Q}\widetilde{R}$ where $\widetilde{A}$ is the matrix $A = QR$ after it has had a number of rows or columns added or deleted.

We presented Fortran codes for a subset of these problems, namely:

- `delcols.f` for updating R when columns have been deleted from A.
- `delcolsq.f` for updating Q when columns have been deleted from A.

- `addcols.f` for updating R when columns have been added to A.
- `addcolsq.f` for updating Q when columns have been added to A.

These codes, and our other algorithms, differ from previous methods as we have exploited the Level 3 BLAS where possible. Also we have dealt with blocks of rows and columns and place no restriction on the dimensions of A.

It was shown that our codes can be much faster than computing the factorization of $\widetilde{A}$ from scratch with existing LAPACK routines. Also, the backward error of our updated factors is comparable to the error bounds of the QR factorization of $\widetilde{A}$.

We propose our codes should be included in a future release of LAPACK, with appropriate testing code to be written.

We have not written Fortran code for updating after adding rows. Also, more investigation is needed to decide on the best approach for a practical code for updating after rows have been deleted.

Appendix A

Code for the Pivoted Cholesky Factorization

A.1 lev2pchol.f

```
      SUBROUTINE LEV2PCHOL( UPLO, N, A, LDA, PIV, RANK, TOL, WORK,
     $                      INFO )
*
*     Modified to include pivoting for semidefinite matrices by
*     Craig Lucas, University of Manchester. January, 2004
*
*     Original LAPACK routine DPOTF2
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     February 29, 1992
*
*     .. Scalar Arguments ..
      DOUBLE PRECISION   TOL
      INTEGER            INFO, LDA, N, RANK
      CHARACTER          UPLO
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), WORK( * )
      INTEGER            PIV( * )
*     ..
*
*  Purpose
*  =======
*
*  LEV2PCHOL computes the Cholesky factorization with complete
```

```
*  pivoting of a real symmetric positive semidefinite matrix A.
*
*  The factorization has the form
*     P' * A * P = U' * U ,  if UPLO = 'U',
*     P' * A * P = L  * L',  if UPLO = 'L',
*  where U is an upper triangular matrix and L is lower triangular, and
*  P is stored as vector PIV.
*
*  This algorithm does not attempt to check that A is positive
*  semidefinite. This version of the algorithm calls level 2 BLAS.
*
*  Arguments
*  =========
*
*  UPLO    (input) CHARACTER*1
*          Specifies whether the upper or lower triangular part of the
*          symmetric matrix A is stored.
*          = 'U':  Upper triangular
*          = 'L':  Lower triangular
*
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the symmetric matrix A.  If UPLO = 'U', the leading
*          n by n upper triangular part of A contains the upper
*          triangular part of the matrix A, and the strictly lower
*          triangular part of A is not referenced.  If UPLO = 'L', the
*          leading n by n lower triangular part of A contains the lower
*          triangular part of the matrix A, and the strictly upper
*          triangular part of A is not referenced.
*
*          On exit, if INFO = 0, the factor U or L from the Cholesky
*          factorization as above. If UPLO = U the first RANK rows
*          of the upper triangular part contains the nonzero part of U.
*          If UPLO = L the first RANK columns of the lower triangular
*          part contains the nonzero part of L. A is unchanged if
*          RANK = 0.
*
*  PIV     (output) INTEGER array, dimension (N)
*          PIV is such that the nonzero entries are P( PIV(K), K ) = 1.
*
*  RANK    (output) INTEGER
```

```
*          The rank of A given by the number of steps the algorithm
*          completed. If the largest algebraic diagonal element is
*          zero than RANK is set to zero.
*
*  TOL     (input) DOUBLE PRECISION
*          User defined tolerance. If TOL < 0, then N*EPS*MAX( A( K,K ) )
*          will be used. The algorithm terminates after K-1 steps if
*          the Kth pivot <= TOL.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*
*  WORK    DOUBLE PRECISION array, dimension (2*N)
*          Work space.
*
*  INFO    (output) INTEGER
*          < 0: if INFO = -K, the K-th argument had an illegal value
*          = 0  algorithm completed successfully.
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER          ( ONE = 1.0D+0, ZERO = 0.0D+0 )
*     ..
*     .. Local Scalars ..
      DOUBLE PRECISION   AJJ, DSTOP, DTEMP, EPS
      INTEGER            ITEMP, J, P, PVT
      LOGICAL            UPPER
*     ..
*     .. External Functions ..
      DOUBLE PRECISION   DLAMCH
      LOGICAL            LSAME
      EXTERNAL           DLAMCH, LSAME
*     ..
*     .. External Subroutines ..
      EXTERNAL           BLAS_DMAX_VAL, DGEMV, DSCAL, DSWAP, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX, SQRT
*     ..
*
*     Test the input parameters
```

```
*
      INFO = 0
      UPPER = LSAME( UPLO, 'U' )
      IF( .NOT.UPPER .AND. .NOT.LSAME( UPLO, 'L' ) ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
         INFO = -4
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'LEV2PC', -INFO )
         RETURN
      END IF
*
*     Quick return if possible
*
      IF( N.EQ.0 )
     $   RETURN
*
*     Initialize PIV
*
      DO 10 P = 1, N
         PIV( P ) = P
   10 CONTINUE
*
*     Get unit roundoff
*
      EPS = DLAMCH( 'E' )
*
*     Compute stopping value
*
      CALL BLAS_DMAX_VAL( N, A( 1, 1 ), LDA+1, PVT, DTEMP )
      AJJ = A( PVT, PVT )
      IF( AJJ.EQ.ZERO ) THEN
         RANK = 0
         GO TO 80
      END IF
*
*     Compute stopping value if not supplied
*
      IF( TOL.LT.ZERO ) THEN
         DSTOP = N*EPS*AJJ
```

```
      ELSE
         DSTOP = TOL
      END IF
*
*     Set first half of WORK to zero, holds dot products
*
      DO 20 P = 1, N
         WORK( P ) = 0
   20 CONTINUE
*
      IF( UPPER ) THEN
*
*        Compute the Cholesky factorization P' * A * P = U' * U
*
         DO 40 J = 1, N
*
*        Find pivot, test for exit, else swap rows and columns
*        Update dot products, compute possible pivots which are
*        stored in the second half of WORK
*
            DO 30 P = J, N
*
               IF( J.GT.1 ) THEN
                  WORK( P ) = WORK( P ) + A( J-1, P )**2
               END IF
               WORK( N+P ) = A( P, P ) - WORK( P )
*
   30       CONTINUE
*
            IF( J.GT.1 ) THEN
               CALL BLAS_DMAX_VAL( N-J+1, WORK( N+J ), 1, ITEMP, DTEMP )
               PVT = ITEMP + J - 1
               AJJ = WORK( N+PVT )
               IF( AJJ.LE.DSTOP ) THEN
                  A( J, J ) = AJJ
                  GO TO 70
               END IF
            END IF
*
            IF( J.NE.PVT ) THEN
*
*              Pivot OK, so can now swap pivot rows and columns
*
```

```
               A( PVT, PVT ) = A( J, J )
               CALL DSWAP( J-1, A( 1, J ), 1, A( 1, PVT ), 1 )
               IF( PVT.LT.N )
     $            CALL DSWAP( N-PVT, A( J, PVT+1 ), LDA,
     $                        A( PVT, PVT+1 ), LDA )
               CALL DSWAP( PVT-J-1, A( J, J+1 ), LDA, A( J+1, PVT ), 1 )
*
*              Swap dot products and PIV
*
               DTEMP = WORK( J )
               WORK( J ) = WORK( PVT )
               WORK( PVT ) = DTEMP
               ITEMP = PIV( PVT )
               PIV( PVT ) = PIV( J )
               PIV( J ) = ITEMP
            END IF
*
            AJJ = SQRT( AJJ )
            A( J, J ) = AJJ
*
*           Compute elements J+1:N of row J
*
            IF( J.LT.N ) THEN
               CALL DGEMV( 'Trans', J-1, N-J, -ONE, A( 1, J+1 ), LDA,
     $                     A( 1, J ), 1, ONE, A( J, J+1 ), LDA )
               CALL DSCAL( N-J, ONE / AJJ, A( J, J+1 ), LDA )
            END IF
*
   40    CONTINUE
*
      ELSE
*
*        Compute the Cholesky factorization P' * A * P = L * L'
*
         DO 60 J = 1, N
*
*        Find pivot, test for exit, else swap rows and columns
*        Update dot products, compute possible pivots which are
*        stored in the second half of WORK
*
            DO 50 P = J, N
*
               IF( J.GT.1 ) THEN
```

```
                  WORK( P ) = WORK( P ) + A( P, J-1 )**2
               END IF
               WORK( N+P ) = A( P, P ) - WORK( P )
*
   50       CONTINUE
*
            IF( J.GT.1 ) THEN
               CALL BLAS_DMAX_VAL( N-J+1, WORK( N+J ), 1, ITEMP, DTEMP )
               PVT = ITEMP + J - 1
               AJJ = WORK( N+PVT )
               IF( AJJ.LE.DSTOP ) THEN
                  A( J, J ) = AJJ
                  GO TO 70
               END IF
            END IF
*
            IF( J.NE.PVT ) THEN
*
*              Pivot OK, so can now swap pivot rows and columns
*
               A( PVT, PVT ) = A( J, J )
               CALL DSWAP( J-1, A( J, 1 ), LDA, A( PVT, 1 ), LDA )
               IF( PVT.LT.N )
     $            CALL DSWAP( N-PVT, A( PVT+1, J ), 1, A( PVT+1, PVT ),
     $                        1 )
               CALL DSWAP( PVT-J-1, A( J+1, J ), 1, A( PVT, J+1 ), LDA )
*
*              Swap dot products and PIV
*
               DTEMP = WORK( J )
               WORK( J ) = WORK( PVT )
               WORK( PVT ) = DTEMP
               ITEMP = PIV( PVT )
               PIV( PVT ) = PIV( J )
               PIV( J ) = ITEMP
            END IF
*
            AJJ = SQRT( AJJ )
            A( J, J ) = AJJ
*
*           Compute elements J+1:N of column J
*
            IF( J.LT.N ) THEN
```

```
               CALL DGEMV( 'No tran', N-J, J-1, -ONE, A( J+1, 1 ), LDA,
     $                     A( J, 1 ), LDA, ONE, A( J+1, J ), 1 )
               CALL DSCAL( N-J, ONE / AJJ, A( J+1, J ), 1 )
            END IF
*
   60    CONTINUE
*
      END IF
*
*     Ran to completion, A has full rank
*
      RANK = N
*
      GO TO 80
   70 CONTINUE
*
*     Rank is number of steps completed
*
      RANK = J - 1
*
   80 CONTINUE
      RETURN
*
*     End of LEV2PCHOL
*
      END
```

A.2 lev3pchol.f

The Level 3 code calls the Level 2 code when the block size is greater than n. The block size is determined by the LAPACK function `ILAENV`. Note we pass the function name `DPOTRF`, the existing Level 3 LAPACK routine for the full rank Cholesky factorization, to return a suitable value.

```
      SUBROUTINE LEV3PCHOL( UPLO, N, A, LDA, PIV, RANK, TOL, WORK,
     $                      INFO )
*
*     Craig Lucas, University of Manchester. January, 2004
*     Some code taken from LAPACK routine DPOTF2
*
*     .. Scalar Arguments ..
      DOUBLE PRECISION   TOL
      INTEGER            INFO, LDA, N, RANK
      CHARACTER          UPLO
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), WORK( * )
      INTEGER            PIV( * )
*     ..
*
*  Purpose
*  =======
*
*  LEV3PCHOL computes the Cholesky factorization with complete
*  pivoting of a real symmetric positive semidefinite matrix A.
*
*  The factorization has the form
*     P' * A * P = U' * U ,  if UPLO = 'U',
*     P' * A * P = L  * L',  if UPLO = 'L',
*  where U is an upper triangular matrix and L is lower triangular, and
*  P is stored as vector PIV.
*
*  This algorithm does not attempt to check that A is positive
*  semidefinite. This version of the algorithm calls level 3 BLAS.
*
*  Arguments
*  =========
*
*  UPLO    (input) CHARACTER*1
*          Specifies whether the upper or lower triangular part of the
*          symmetric matrix A is stored.
```

```
*          = 'U':  Upper triangular
*          = 'L':  Lower triangular
*
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the symmetric matrix A.  If UPLO = 'U', the leading
*          n by n upper triangular part of A contains the upper
*          triangular part of the matrix A, and the strictly lower
*          triangular part of A is not referenced.  If UPLO = 'L', the
*          leading n by n lower triangular part of A contains the lower
*          triangular part of the matrix A, and the strictly upper
*          triangular part of A is not referenced.
*
*          On exit, if INFO = 0, the factor U or L from the Cholesky
*          factorization as above. If UPLO = U the first RANK rows
*          of the upper triangular part contains the nonzero part of U.
*          If UPLO = L the first RANK columns of the lower triangular
*          part contains the nonzero part of L. A is unchanged if
*          RANK = 0.
*
*  PIV     (output) INTEGER array, dimension (N)
*          PIV is such that the nonzero entries are P( PIV(K), K ) = 1.
*
*  RANK    (output) INTEGER
*          The rank of A given by the number of steps the algorithm
*          completed. If the largest algebraic diagonal element is
*          zero than RANK is set to zero.
*
*  TOL     (input) DOUBLE PRECISION
*          User defined tolerance. If TOL < 0, then N*EPS*MAX( A(K,K) )
*          will be used. The algorithm terminates after K-1 steps if
*          the Kth pivot <= TOL.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*
*  WORK    DOUBLE PRECISION array, dimension (2*N)
*          Work space.
*
*  INFO    (output) INTEGER
*          < 0: if INFO = -K, the K-th argument had an illegal value
```

```
*          = 0  algorithm completed successfully.
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER          ( ONE = 1.0D+0, ZERO = 0.0D+0 )
*     ..
*     .. Local Scalars ..
      DOUBLE PRECISION   AJJ, DSTOP, DTEMP, EPS
      INTEGER            ITEMP, J, JB, K, NB, P, PVT
      LOGICAL            UPPER
*     ..
*     .. External Functions ..
      REAL               DLAMCH
      INTEGER            ILAENV
      LOGICAL            LSAME
      EXTERNAL           DLAMCH, ILAENV, LSAME
*     ..
*     .. External Subroutines ..
      EXTERNAL           BLAS_DMAX_VAL, DGEMV, DSCAL, DSWAP, DSYRK,
     $                   LEV2PCHOL, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX, MIN, SQRT
*     ..
*
*     Test the input parameters.
*
      INFO = 0
      UPPER = LSAME( UPLO, 'U' )
      IF( .NOT.UPPER .AND. .NOT.LSAME( UPLO, 'L' ) ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
         INFO = -4
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'LEV3PC', -INFO )
         RETURN
      END IF
*
```

```
*     Quick return if possible
*
      IF( N.EQ.0 )
     $   RETURN
*
*     Get block size
*
      NB = ILAENV( 1, 'DPOTRF', UPLO, N, -1, -1, -1 )
      IF( NB.LE.1 .OR. NB.GE.N ) THEN
*
*        Use unblocked code
*
         CALL LEV2PCHOL( UPLO, N, A( 1, 1 ), LDA, PIV, RANK, TOL, WORK,
     $                   INFO )
         GO TO 110
*
      ELSE
*
*     Initialize PIV
*
         DO 10 P = 1, N
            PIV( P ) = P
   10    CONTINUE
*
*     Get unit roundoff
*
         EPS = DLAMCH( 'E' )
*
*     Compute stopping value
*
         CALL BLAS_DMAX_VAL( N, A( 1, 1 ), LDA+1, PVT, DTEMP )
         AJJ = A( PVT, PVT )
         IF( AJJ.EQ.ZERO ) THEN
            RANK = 0
            GO TO 110
         END IF
*
*     Compute stopping value if not supplied
*
         IF( TOL.LT.ZERO ) THEN
            DSTOP = N*EPS*AJJ
         ELSE
            DSTOP = TOL
```

```
         END IF
*
*
         IF( UPPER ) THEN
*
*           Compute the Cholesky factorization P' * A * P = U' * U
*
            DO 50 K = 1, N, NB
*
*              Account for last block not being NB wide
*
               JB = MIN( NB, N-K+1 )
*
*              Set relevant part of first half of WORK to zero,
*              holds dot products
*
               DO 20 P = K, N
                  WORK( P ) = 0
   20          CONTINUE
*
               DO 40 J = K, K + JB - 1
*
*              Find pivot, test for exit, else swap rows and columns
*              Update dot products, compute possible pivots which are
*              stored in the second half of WORK
*
                  DO 30 P = J, N
*
                     IF( J.GT.K ) THEN
                        WORK( P ) = WORK( P ) + A( J-1, P )**2
                     END IF
                     WORK( N+P ) = A( P, P ) - WORK( P )
*
   30             CONTINUE
*
                  IF( J.GT.1 ) THEN
                     CALL BLAS_DMAX_VAL( N-J+1, WORK( N+J ), 1, ITEMP,
     $                                   DTEMP )
                     PVT = ITEMP + J - 1
                     AJJ = WORK( N+PVT )
                     IF( AJJ.LE.DSTOP ) THEN
                        A( J, J ) = AJJ
                        GO TO 100
```

```
                     END IF
                  END IF
*
                  IF( J.NE.PVT ) THEN
*
*                    Pivot OK, so can now swap pivot rows and columns
*
                     A( PVT, PVT ) = A( J, J )
                     CALL DSWAP( J-1, A( 1, J ), 1, A( 1, PVT ), 1 )
                     IF( PVT.LT.N )
     $                  CALL DSWAP( N-PVT, A( J, PVT+1 ), LDA,
     $                              A( PVT, PVT+1 ), LDA )
                     CALL DSWAP( PVT-J-1, A( J, J+1 ), LDA,
     $                           A( J+1, PVT ), 1 )
*
*                    Swap dot products and PIV
*
                     DTEMP = WORK( J )
                     WORK( J ) = WORK( PVT )
                     WORK( PVT ) = DTEMP
                     ITEMP = PIV( PVT )
                     PIV( PVT ) = PIV( J )
                     PIV( J ) = ITEMP
                  END IF
*
                  AJJ = SQRT( AJJ )
                  A( J, J ) = AJJ
*
*                 Compute elements J+1:N of row J.
*
                  IF( J.LT.N ) THEN
                     CALL DGEMV( 'Trans', J-K, N-J, -ONE, A( K, J+1 ),
     $                           LDA, A( K, J ), 1, ONE, A( J, J+1 ),
     $                           LDA )
                     CALL DSCAL( N-J, ONE / AJJ, A( J, J+1 ), LDA )
                  END IF
*
   40          CONTINUE
*
*              Update trailing matrix, J already incremented
*
               IF( K+JB.LE.N ) THEN
                  CALL DSYRK( 'Upper', 'Trans', N-J+1, JB, -ONE,
```

```
     $                        A( K, J ), LDA, ONE, A( J, J ), LDA )
               END IF
*
   50       CONTINUE
*
         ELSE
*
*        Compute the Cholesky factorization P' * A * P = L * L'
*
            DO 90 K = 1, N, NB
*
*              Account for last block not being NB wide
*
               JB = MIN( NB, N-K+1 )
*
*              Set relevant part of first half of WORK to zero,
*              holds dot products
*
               DO 60 P = K, N
                  WORK( P ) = 0
   60          CONTINUE
*
               DO 80 J = K, K + JB - 1
*
*              Find pivot, test for exit, else swap rows and columns
*              Update dot products, compute possible pivots which are
*              stored in the second half of WORK
*
                  DO 70 P = J, N
*
                     IF( J.GT.K ) THEN
                        WORK( P ) = WORK( P ) + A( P, J-1 )**2
                     END IF
                     WORK( N+P ) = A( P, P ) - WORK( P )
*
   70             CONTINUE
*
                  IF( J.GT.1 ) THEN
                     CALL BLAS_DMAX_VAL( N-J+1, WORK( N+J ), 1, ITEMP,
     $                                   DTEMP )
                     PVT = ITEMP + J - 1
                     AJJ = WORK( N+PVT )
                     IF( AJJ.LE.DSTOP ) THEN
```

```
                     A( J, J ) = AJJ
                     GO TO 100
                  END IF
               END IF
*
               IF( J.NE.PVT ) THEN
*
*                 Pivot OK, so can now swap pivot rows and columns
*
                  A( PVT, PVT ) = A( J, J )
                  CALL DSWAP( J-1, A( J, 1 ), LDA, A( PVT, 1 ), LDA )
                  IF( PVT.LT.N )
     $               CALL DSWAP( N-PVT, A( PVT+1, J ), 1,
     $                           A( PVT+1, PVT ), 1 )
                  CALL DSWAP( PVT-J-1, A( J+1, J ), 1, A( PVT, J+1 ),
     $                        LDA )
*
*                 Swap dot products and PIV
*
                  DTEMP = WORK( J )
                  WORK( J ) = WORK( PVT )
                  WORK( PVT ) = DTEMP
                  ITEMP = PIV( PVT )
                  PIV( PVT ) = PIV( J )
                  PIV( J ) = ITEMP
               END IF
*
               AJJ = SQRT( AJJ )
               A( J, J ) = AJJ
*
*              Compute elements J+1:N of column J.
*
               IF( J.LT.N ) THEN
                  CALL DGEMV( 'No tran', N-J, J-K, -ONE, A( J+1, K ),
     $                        LDA, A( J, K ), LDA, ONE, A( J+1, J ),
     $                        1 )
                  CALL DSCAL( N-J, ONE / AJJ, A( J+1, J ), 1 )
               END IF
*
   80       CONTINUE
*
*           Update trailing matrix, J already incremented
*
```

```
               IF( K+JB.LE.N ) THEN
                  CALL DSYRK( 'Lower', 'No Trans', N-J+1, JB, -ONE,
     $                        A( J, K ), LDA, ONE, A( J, J ), LDA )
               END IF
*
   90       CONTINUE
*
         END IF
      END IF
*
*     Ran to completion, A has full rank
*
      RANK = N
*
      GO TO 110
  100 CONTINUE
*
*     Rank is the number of steps completed
*
      RANK = J - 1
*
  110 CONTINUE
      RETURN
*
*     End of LEV3PCHOL
*
      END
```

A.3 blas_dmax_val.f

The subroutine BLAS_DMAX_VAL is modified from the BLAS function IDAMAX to return the largest algebraic value of a vector and the smallest index that contains that value. This routine is used as there is not an appropriate routine in the current version of the BLAS. IDAMAX returns the largest *absolute* value and its index. The name and interface conform to the BLAS Technical Forum Standard [46], which gives details of a future release of the BLAS.

```
      SUBROUTINE BLAS_DMAX_VAL( N, X, INCX, K, R )
*
*     BLAS_DMAX_VAL finds the largest component of X, R, and
*     determines the smallest index, K, such that X(K) = R.
*     Craig Lucas, University of Manchester. June, 2003
*
*     Modified from the BLAS function IDAMAX:
*     Jack Dongarra, LINPACK, 3/11/78.
*
*     .. Scalar Arguments ..
      DOUBLE PRECISION   R
      INTEGER            INCX, K, N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   X( * )
*     ..
*     .. Local Scalars ..
      INTEGER            I, IX
*     ..
      K = 0
      IF( N.LT.1 .OR. INCX.LE.0 )
     $   RETURN
      K = 1
      IF( N.EQ.1 )
     $   RETURN
      IF( INCX.EQ.1 )
     $   GO TO 30
*
*     Code for increment not equal to 1
*
      IX = 1
      R = X( 1 )
      IX = IX + INCX
      DO 20 I = 2, N
         IF( X( IX ).LE.R )
```

```
     $      GO TO 10
            K = I
            R = X( IX )
   10       CONTINUE
            IX = IX + INCX
   20    CONTINUE
         RETURN
*
*        Code for increment equal to 1
*
   30    CONTINUE
         R = X( 1 )
         DO 40 I = 2, N
            IF( X( I ).LE.R )
     $      GO TO 40
            K = I
            R = X( I )
   40    CONTINUE
         RETURN
         END
```

Appendix B

Testing Code for the Cholesky Routines

CL_DCHKAA runs the test code for LEV2PCHOL and LEV3PCHOL. It reads in user set parameters and calls the subroutine CL_DCHKPO. It is based on the LAPACK testing routine DCHKAA. Other dependent routines follow the same naming convention of prefixing CL_ where existing LAPACK routines have been adapted for our code.

CL_DCHKPO loops through all the different cases specified in the input file. Test matrices are generated, input arguments are checked and the backward error of the computed factor is measured. Full details are given in the individual files.

CL_DCHKPO calls the following adapted LAPACK testing routines not included in this appendix:

- CL_DERRPO checks LEV2PCHOL and LEV3PCHOL exit correctly when input parameters are incorrect.
- CL_DLATB4 sets parameters for the following routine.
- CL_DLATMS generates random matrices. Requires CL_DLATM1 which specifies eigenvalues of the random matrix.
- CL_DPOT01 calculates the backward error of the computed Cholesky factors.
- CL_ALAERH handles the output from the testing routines. CL_ALAHD is required which prints details to the screen.

The following LAPACK testing routines are also required for setting parameters to other routines, handling errors and printing results:

```
ALADHD     ALAESM     ALAREQ     ALASUM     CHKXER     XLAENV
```

Special versions of the LAPACK routines `ILAENV`, which sets some of the input parameters to `LEV2PCHOL` and `LEV3PCHOL`, and `XERBLA` which handles errors are also required.

The full set of testing files can be found at `http://www.ma.man.ac.uk/~clucas/cholesky`.

B.1 cl_dchkaa.f

```
      PROGRAM CL_DCHKAA
*
*     Craig Lucas, University of Manchester. March, 2004.
*     Based on the following LAPACK routine.
*
*     DCHKAA
*  -- LAPACK test routine (version 3.0) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     June 30, 1999
*
*  Purpose
*  =======
*
*  CL_DCHKAA is a test program for the DOUBLE PRECISION routine
*  LEV3PCHOL, calling LEV2PCHOL for small N.
*
*  The program must be driven by a short data file. The first 9 records
*  specify problem dimensions and program options using list-directed
*  input.  The last line specify the LAPACK test path and the
*  number of matrix types to use in testing. The test path is a legacy
*  of the original LAPACK code and is always 'DPO'. An annotated example
*  of a data file can be obtained by deleting the first 3 characters
*  from the following 11 lines:
*  Data file for testing DOUBLE PRECISION routine LEV3PCHOL
*  6                      Number of values of N
*  0 10 20 30 100 200     Values of N
*  3                      Number of values of NB
*  10 20  50              Values of NB (the block size)
```

```
*  3                      Number of values of RANK
*  30 50 90               Values of rank (as a % of N)
*  1                      Threshold value of test ratio
*  T                      Put T to test the LAPACK routines (T = true)
*  T                      Put T to test the error exits
*  DPO    9               List types on next line if 0 < NTYPES <  9
*
*  Internal Parameters
*  ===================
*
*  NMAX    INTEGER
*          The maximum allowable value for N
*
*  MAXIN   INTEGER
*          The number of different values that can be used for each of
*          N, NB, and RANK
*
*  NIN     INTEGER
*          The unit number for input
*
*  NOUT    INTEGER
*          The unit number for output
*
*  =====================================================================
*
*     .. Parameters ..
      INTEGER            NMAX
      PARAMETER          ( NMAX = 1000 )
      INTEGER            MAXIN
      PARAMETER          ( MAXIN = 12 )
      INTEGER            MATMAX
      PARAMETER          ( MATMAX = 9 )
      INTEGER            NIN, NOUT
      PARAMETER          ( NIN = 5, NOUT = 6 )
*     ..
*     .. Local Scalars ..
      DOUBLE PRECISION   EPS, S1, S2, THRESH
      INTEGER            I, IC, J, K, LDA, NB, NMATS, NN, NNB, NNB2,
     $                   NRANK, NTYPES
      LOGICAL            FATAL, TSTCHK, TSTERR
      CHARACTER          C1
      CHARACTER*2        C2
      CHARACTER*3        PATH
```

```
      CHARACTER*10       INTSTR
      CHARACTER*72       ALINE
*     ..
*     .. Local Arrays ..
*
      DOUBLE PRECISION   A( NMAX*NMAX, 3 ), PIV( NMAX ),
     $                   RWORK( NMAX ), WORK( NMAX*3 )
      INTEGER            IWORK( 25*NMAX ), NBVAL( MAXIN ),
     $                   NBVAL2( MAXIN ), NVAL( MAXIN ),
     $                   RANKVAL( MAXIN )
      LOGICAL            DOTYPE( MATMAX )
*     ..
*     .. External Functions ..
      DOUBLE PRECISION   DLAMCH, DSECND
      LOGICAL            LSAME, LSAMEN
      EXTERNAL           DLAMCH, DSECND, LSAME, LSAMEN
*     ..
*     .. External Subroutines ..
*
      EXTERNAL           ALAREQ, CL_DCHKPO
*     ..
*     .. Scalars in Common ..
      INTEGER            INFOT, NUNIT
      LOGICAL            LERR, OK
      CHARACTER*6        SRNAMT
*     ..
*     .. Arrays in Common ..
      INTEGER            IPARMS( 100 )
*     ..
*     .. Common blocks ..
      COMMON             / CLAENV / IPARMS
      COMMON             / INFOC / INFOT, NUNIT, OK, LERR
      COMMON             / SRNAMC / SRNAMT
*     ..
*     .. Data statements ..
      DATA               INTSTR / '0123456789' /
*     ..
*
      S1 = DSECND( )
      LDA = NMAX
      FATAL = .FALSE.
*
*     Read a dummy line.
```

```
*
      READ( NIN, FMT = * )
*
*     Report values of parameters.
*
      WRITE( NOUT, FMT = 9994 )
*
*     Read the values of N
*
      READ( NIN, FMT = * )NN
      IF( NN.LT.1 ) THEN
         WRITE( NOUT, FMT = 9996 )' NN ', NN, 1
         NN = 0
         FATAL = .TRUE.
      ELSE IF( NN.GT.MAXIN ) THEN
         WRITE( NOUT, FMT = 9995 )' NN ', NN, MAXIN
         NN = 0
         FATAL = .TRUE.
      END IF
      READ( NIN, FMT = * )( NVAL( I ), I = 1, NN )
      DO 10 I = 1, NN
         IF( NVAL( I ).LT.0 ) THEN
            WRITE( NOUT, FMT = 9996 )' N  ', NVAL( I ), 0
            FATAL = .TRUE.
         ELSE IF( NVAL( I ).GT.NMAX ) THEN
            WRITE( NOUT, FMT = 9995 )' N  ', NVAL( I ), NMAX
            FATAL = .TRUE.
         END IF
   10 CONTINUE
      IF( NN.GT.0 )
     $   WRITE( NOUT, FMT = 9993 )'N          ',
     $   ( NVAL( I ), I = 1, NN )
*
*     Read the values of NB
*
      READ( NIN, FMT = * )NNB
      IF( NNB.LT.1 ) THEN
         WRITE( NOUT, FMT = 9996 )'NNB ', NNB, 1
         NNB = 0
         FATAL = .TRUE.
      ELSE IF( NNB.GT.MAXIN ) THEN
         WRITE( NOUT, FMT = 9995 )'NNB ', NNB, MAXIN
         NNB = 0
```

```
         FATAL = .TRUE.
      END IF
      READ( NIN, FMT = * )( NBVAL( I ), I = 1, NNB )
      DO 20 I = 1, NNB
         IF( NBVAL( I ).LT.0 ) THEN
            WRITE( NOUT, FMT = 9996 )' NB ', NBVAL( I ), 0
            FATAL = .TRUE.
         END IF
   20 CONTINUE
      IF( NNB.GT.0 )
     $   WRITE( NOUT, FMT = 9993 )' NB         ',
     $   ( NBVAL( I ), I = 1, NNB )
*
*     Set NBVAL2 to be the set of unique values of NB
*
      NNB2 = 0
      DO 40 I = 1, NNB
         NB = NBVAL( I )
         DO 30 J = 1, NNB2
            IF( NB.EQ.NBVAL2( J ) )
     $         GO TO 40
   30    CONTINUE
         NNB2 = NNB2 + 1
         NBVAL2( NNB2 ) = NB
   40 CONTINUE
*
*     Read the values of RANKVAL
*
      READ( NIN, FMT = * )NRANK
      IF( NN.LT.1 ) THEN
         WRITE( NOUT, FMT = 9996 )' NRANK ', NRANK, 1
         NRANK = 0
         FATAL = .TRUE.
      ELSE IF( NN.GT.MAXIN ) THEN
         WRITE( NOUT, FMT = 9995 )' NRANK ', NRANK, MAXIN
         NRANK = 0
         FATAL = .TRUE.
      END IF
      READ( NIN, FMT = * )( RANKVAL( I ), I = 1, NRANK )
      DO 50 I = 1, NRANK
         IF( RANKVAL( I ).LT.0 ) THEN
            WRITE( NOUT, FMT = 9996 )' RANK  ', RANKVAL( I ), 0
            FATAL = .TRUE.
```

```
         ELSE IF( RANKVAL( I ).GT.100 ) THEN
            WRITE( NOUT, FMT = 9995 )' RANK  ', RANKVAL( I ), 100
            FATAL = .TRUE.
         END IF
   50 CONTINUE
      IF( NRANK.GT.0 )
     $   WRITE( NOUT, FMT = 9993 )'RANK % OF N',
     $   ( RANKVAL( I ), I = 1, NRANK )
*
*     Read the threshold value for the test ratios.
*
      READ( NIN, FMT = * )THRESH
      WRITE( NOUT, FMT = 9992 )THRESH
*
*     Read the flag that indicates whether to test the routine.
*
      READ( NIN, FMT = * )TSTCHK
*
*     Read the flag that indicates whether to test the error exits.
*
      READ( NIN, FMT = * )TSTERR
*
      IF( FATAL ) THEN
         WRITE( NOUT, FMT = 9999 )
         STOP
      END IF
*
*     Calculate and print the machine dependent constants.
*
      EPS = DLAMCH( 'Underflow threshold' )
      WRITE( NOUT, FMT = 9991 )'underflow', EPS
      EPS = DLAMCH( 'Overflow threshold' )
      WRITE( NOUT, FMT = 9991 )'overflow ', EPS
      EPS = DLAMCH( 'Epsilon' )
      WRITE( NOUT, FMT = 9991 )'precision', EPS
      WRITE( NOUT, FMT = * )
*
*     Read a test path and the number of matrix types to use.
*
      READ( NIN, FMT = '(A72)', END = 110 )ALINE
      PATH = ALINE( 1: 3 )
      NMATS = MATMAX
      I = 3
```

```
   60 CONTINUE
      I = I + 1
      IF( I.GT.72 ) THEN
         NMATS = MATMAX
         GO TO 100
      END IF
      IF( ALINE( I: I ).EQ.' ' )
     $   GO TO 60
      NMATS = 0
   70 CONTINUE
      C1 = ALINE( I: I )
      DO 80 K = 1, 10
         IF( C1.EQ.INTSTR( K: K ) ) THEN
            IC = K - 1
            GO TO 90
         END IF
   80 CONTINUE
      GO TO 100
   90 CONTINUE
      NMATS = NMATS*10 + IC
      I = I + 1
      IF( I.GT.72 )
     $   GO TO 100
      GO TO 70
  100 CONTINUE
      C1 = PATH( 1: 1 )
      C2 = PATH( 2: 3 )
*
*     Check first character for correct precision.
*
      IF( .NOT.LSAME( C1, 'Double precision' ) ) THEN
         WRITE( NOUT, FMT = 9990 )PATH
*
      ELSE IF( NMATS.LE.0 ) THEN
*
*        Check for a positive number of tests requested.
*
         WRITE( NOUT, FMT = 9989 )PATH
*
      ELSE IF( LSAMEN( 2, C2, 'PO' ) ) THEN
*
*        PO:  positive semi-definite matrices
*
```

```
         NTYPES = 9
*
         CALL ALAREQ( PATH, NMATS, DOTYPE, NTYPES, NIN, NOUT )
*
         IF( TSTCHK ) THEN
            CALL CL_DCHKPO( DOTYPE, NN, NVAL, NNB2, NBVAL2, NRANK,
     $                      RANKVAL,THRESH, TSTERR, LDA, A( 1, 1 ),
     $                      A( 1, 2 ),A( 1, 3 ), PIV, WORK, RWORK,
     $                      IWORK, NOUT )
         ELSE
            WRITE( NOUT, FMT = 9989 )PATH
         END IF
*
      ELSE
*
         WRITE( NOUT, FMT = 9990 )PATH
      END IF
*
*     Branch to this line when the last record is read.
*
  110 CONTINUE
      CLOSE ( NIN )
      S2 = DSECND( )
      WRITE( NOUT, FMT = 9998 )
      WRITE( NOUT, FMT = 9997 )S2 - S1
*
 9999 FORMAT( / ' Execution not attempted due to input errors' )
 9998 FORMAT( / ' End of tests' )
 9997 FORMAT( ' Total time used = ', F12.2, ' seconds', / )
 9996 FORMAT( ' Invalid input value: ', A5, '=', I6, '; must be >=',
     $      I6 )
 9995 FORMAT( ' Invalid input value: ', A5, '=', I6, '; must be <=',
     $      I6 )
 9994 FORMAT( ' Tests of the DOUBLE PRECISION routine LEV3PCHOL',
     $      / ' LAPACK VERSION X.X, ', / /
     $      ' The following parameter values will be used:' )
 9993 FORMAT( 4X, A12, ':  ', 10I6, / 11X, 10I6 )
 9992 FORMAT( / ' Routines pass computational tests if test ratio is ',
     $      'less than', F8.2, / )
 9991 FORMAT( ' Relative machine ', A, ' is taken to be', D16.6 )
 9990 FORMAT( / 1X, A3, ':  Unrecognized path name' )
 9989 FORMAT( / 1X, A3, ' routines were not tested' )
 9988 FORMAT( / 1X, A3, ' driver routines were not tested' )
```

```
*
*     End of CL_DCHKAA
*
      END
```

B.2 cl_dchkpo.f

```
      SUBROUTINE CL_DCHKPO( DOTYPE, NN, NVAL, NNB, NBVAL, NRANK,
     $                      RANKVAL, THRESH, TSTERR, NMAX, A, AFAC,
     $                      PERM, PIV, WORK, RWORK, IWORK, NOUT )
*
*     Craig Lucas, University of Manchester. March, 2004.
*     Based on the following LAPACK routine. Arguments added are
*     NRANK, RANKVAL, PERM and PIV
*
*     DCHKPO
*  -- LAPACK test routine (version 3.0) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     December 7, 1999
*
*     .. Scalar Arguments ..
      DOUBLE PRECISION   THRESH
      REAL               PIV
      INTEGER            NMAX, NN, NNB, NOUT, NRANK
      LOGICAL            TSTERR
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( * ), AFAC( * ), PERM( * ), RWORK( * ),
     $                   WORK( * )
      INTEGER            IWORK( * ), NBVAL( * ), NVAL( * ), RANKVAL( * )
      LOGICAL            DOTYPE( * )
*     ..
*
*  Purpose
*  =======
*
*  CL_DCHKPO tests LEV3PCHOL.
*
*  Arguments
*  =========
*
*  DOTYPE  (input) LOGICAL array, dimension (NTYPES)
*          The matrix types to be used for testing.  Matrices of type j
*          (for 1 <= j <= NTYPES) are used for testing if DOTYPE(j) =
*          .TRUE.; if DOTYPE(j) = .FALSE., then type j is not used.
*
*  NN      (input) INTEGER
```

```
*          The number of values of N contained in the vector NVAL.
*
*  NVAL    (input) INTEGER array, dimension (NN)
*          The values of the matrix dimension N.
*
*  NNB     (input) INTEGER
*          The number of values of NB contained in the vector NBVAL.
*
*  NBVAL   (input) INTEGER array, dimension (NBVAL)
*          The values of the block size NB.
*
*  NRANK   (input) INTEGER
*          The number of values of RANK contained in the vector RANKVAL.
*
*  RANKVAL (input) INTEGER array, dimension (NBVAL)
*          The values of the block size NB.
*
*  THRESH  (input) DOUBLE PRECISION
*          The threshold value for the test ratios.  A result is
*          included in the output file if RESULT >= THRESH.  To have
*          every test ratio printed, use THRESH = 0.
*
*  TSTERR  (input) LOGICAL
*          Flag that indicates whether error exits are to be tested.
*
*  NMAX    (input) INTEGER
*          The maximum value permitted for N, used in dimensioning the
*          work arrays.
*
*  A       (workspace) DOUBLE PRECISION array, dimension (NMAX*NMAX)
*
*  AFAC    (workspace) DOUBLE PRECISION array, dimension (NMAX*NMAX)
*
*  PERM    (workspace) DOUBLE PRECISION array, dimension (NMAX*NMAX)
*
*  PIV     (workspace) INTEGER array, dimension (NMAX)
*
*  WORK    (workspace) DOUBLE PRECISION array, dimension (NMAX*3)
*
*  RWORK   (workspace) DOUBLE PRECISION array, dimension (NMAX)
*
*  IWORK   (workspace) INTEGER array, dimension (NMAX)
*
```

```
*  NOUT    (input) INTEGER
*          The unit number for output.
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE
      PARAMETER          ( ONE = 1.0D+0 )
      INTEGER            NTYPES
      PARAMETER          ( NTYPES = 9 )
*     ..
*     .. Local Scalars ..
      DOUBLE PRECISION   ANORM, CNDNUM, RESULT, TOL
      INTEGER            COMPRANK, I, IMAT, IN, INB, INFO, IRANK, IUPLO,
     $                   IZERO, KL, KU, LDA, MODE, N, NB, NERRS, NFAIL,
     $                   NIMAT, NRUN, RANK, RANKDIFF
      CHARACTER          DIST, TYPE, UPLO, XTYPE
      CHARACTER*3        PATH
*     ..
*     .. Local Arrays ..
      INTEGER            ISEED( 4 ), ISEEDY( 4 )
      CHARACTER          UPLOS( 2 )
*     ..
*     .. External Subroutines ..
      EXTERNAL           CL_ALAERH, CL_ALAHD, ALASUM, CL_DERRPO,
     $                   DLACPY, CL_DLATB4, CL_DLATMS, CL_DPOT01,
     $                   LEV3PCHOL, XLAENV
*     ..
*     .. Scalars in Common ..
      INTEGER            INFOT, NUNIT
      LOGICAL            LERR, OK
      CHARACTER*6        SRNAMT
*     ..
*     .. Common blocks ..
      COMMON             / INFOC / INFOT, NUNIT, OK, LERR
      COMMON             / SRNAMC / SRNAMT
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          DBLE, MAX
*     ..
*     .. Data statements ..
      DATA               ISEEDY / 1988, 1989, 1990, 1991 /
      DATA               UPLOS / 'U', 'L' /
```

```
*     ..
*
*     Initialize constants and the random number seed.
*
      PATH( 1: 1 ) = 'Double precision'
      PATH( 2: 3 ) = 'PO'
      NRUN = 0
      NFAIL = 0
      NERRS = 0
      DO 10 I = 1, 4
         ISEED( I ) = ISEEDY( I )
   10 CONTINUE
*
*     Test the error exits
*
      IF( TSTERR )
     $   CALL CL_DERRPO( PATH, NOUT )
      INFOT = 0
      CALL XLAENV( 2, 2 )
*
*     Do for each value of N in NVAL
*
      DO 60 IN = 1, NN
         N = NVAL( IN )
         LDA = MAX( N, 1 )
         XTYPE = 'N'
         NIMAT = NTYPES
         IF( N.LE.0 )
     $      NIMAT = 1
*
         IZERO = 0
         DO 50 IMAT = 1, NIMAT
*
*           Do the tests only if DOTYPE( IMAT ) is true.
*
            IF( .NOT.DOTYPE( IMAT ) )
     $         GO TO 50
*
*              Do for each value of RANK in RANKVAL
*
            DO 40 IRANK = 1, NRANK
*
*              Only repeat test 3 to 5 for different ranks
```

```
*              Other tests use full rank
*
               IF( ( IMAT.LT.3 .OR. IMAT.GT.5 ) .AND. IRANK.GT.1 )
     $            GO TO 40
*
               RANK = ( DBLE( RANKVAL( IRANK ) ) / 100.D+0 )*N
*
*
*           Do first for UPLO = 'U', then for UPLO = 'L'
*
               DO 30 IUPLO = 1, 2
                  UPLO = UPLOS( IUPLO )
*
*              Set up parameters with DLATB4 and generate a test matrix
*              with DLATMS.
*
                  CALL CL_DLATB4( PATH, IMAT, N, N, TYPE, KL, KU, ANORM,
     $                            MODE, CNDNUM, DIST )
*
                  SRNAMT = 'DLATMS'
                  CALL CL_DLATMS( N, N, DIST, ISEED, TYPE, RWORK, MODE,
     $                            CNDNUM, ANORM, RANK, KL, KU, UPLO, A,
     $                            LDA, WORK, INFO )
*
*              Check error code from DLATMS.
*
                  IF( INFO.NE.0 ) THEN
                     CALL CL_ALAERH( PATH, 'DLATMS', INFO, 0, UPLO, N,
     $                               N, -1, -1, -1, IMAT, NFAIL, NERRS,
     $                               NOUT )
                     GO TO 30
                  END IF
*
*              Do for each value of NB in NBVAL
*
                  DO 20 INB = 1, NNB
                     NB = NBVAL( INB )
                     CALL XLAENV( 1, NB )
*
*                 Compute the pivoted L*L' or U'*U factorization
*                 of the matrix.
*
                     CALL DLACPY( UPLO, N, N, A, LDA, AFAC, LDA )
```

```
                  SRNAMT = 'LEV3PC'
*
*                 Use default tolerance
*
                  TOL = -ONE
                  CALL LEV3PCHOL( UPLO, N, AFAC, LDA, PIV, COMPRANK,
     $                            TOL, WORK, INFO )
*
*                 Check error code from LEV3PCHOL.
*
                  IF( INFO.NE.IZERO ) THEN
                     CALL CL_ALAERH( PATH, 'LEV3PC', INFO, IZERO,
     $                               UPLO,N, N, -1, -1, NB, IMAT,
     $                               NFAIL, NERRS, NOUT )
                     GO TO 20
                  END IF
*
*                 Skip the test if INFO is not 0.
*
                  IF( INFO.NE.0 )
     $               GO TO 20
*
*                 Reconstruct matrix from factors and compute residual.
*
*                 PERM holds permuted L*L^T or U^T*U
*
                  CALL CL_DPOT01( UPLO, N, A, LDA, AFAC, LDA, PERM,
     $                            LDA, PIV, RWORK, RESULT, COMPRANK )
*
*                 Print information about the tests that did not pass
*                 the threshold or where computed rank was not RANK.
*
                  RANKDIFF = RANK - COMPRANK
                  IF( N.EQ.0 )
     $               RANKDIFF = 0
                  IF( RESULT.GE.THRESH .OR. RANKDIFF.NE.0 ) THEN
                     IF( NFAIL.EQ.0 .AND. NERRS.EQ.0 )
     $                  CALL CL_ALAHD( NOUT, PATH )
                     WRITE( NOUT, FMT = 9999 )UPLO, N, RANK,
     $                  RANKDIFF, NB, IMAT, RESULT
                     NFAIL = NFAIL + 1
                  END IF
                  NRUN = NRUN + 1
```

```
   20             CONTINUE
*
   30          CONTINUE
   40       CONTINUE
   50    CONTINUE
   60 CONTINUE
*
*     Print a summary of the results.
*
      CALL ALASUM( PATH, NOUT, NFAIL, NRUN, NERRS )
*
 9999 FORMAT( ' UPLO = ''', A1, ''', N =', I5, ', RANK =', I3,
     $      ', Diff =', I5, ', NB =', I4, ', type ', I2, ', Ratio =',
     $      G12.5 )
      RETURN
*
*     End of CL_DCHKPO
*
      END
```

Appendix C

MATLAB Code for the Symmetric Semidefinite Generalized Eigenvalue Problem

C.1 ssgep.m

```
function [U,D,r] = ssgep(A,B,rrd1,rrd3,rrd4,reltol,normtol)
%SSGEP  Solves the symmetric semidefinite generalized eigenvalue problem.
%       This function returns the finite eigenvalues and the
%       eigenvectors of the symmetric semidefinite eigenvalue problem
%       Ax = \lambda Bx, A = A', B = B' and B semidefinite.
%
%       [U,D,r] = ssgep(A,B,rrd1,rrd3,rrd4,reltol,normtol)
%       U is a matrix of eigenvectors, the first r(6) columns contain
%       those corresponding to finite eigenvalues, the remaining r(3) +
%       r(5) correspond to infinite eigenvalues. The diagonal elements
%       of D are the r(6) finite eigenvalues. r(1) is the numerical
%       rank of B. r(7) is the dimension of the singular part of the
%       pencil which has been deflated out of the problem.
%
%       rrd1 specifies rank revealing decomposition (RRD) for Step 1
%       'spec' spectral decomposition via eig.m (default)
%       'chol' Cholesky decomposition via cholp.m
%
%       rrd3 specifies RRD for Step 3
%       'spec' as above (default)
%       'ldlt' LDL' factorization via lqdqtlt.m
%
%       rrd4 specifies RRD for Step 4
```

```
%       'svd' singular value decomposition via svd.m (default)
%       'qrp' column pivoted QR decomposition via qr.m
%       'qr' QR factorization via qr.m, fails for nonregular pencils
%       'cod' Complete orthogonal decomposition via cod.m
%
%       reltol is a tolerance for defining the rank of an approximate
%       diagonal matrix. For an RRD C = XDZ of dimension m-by-n with the
%       elements in D ordered in descending order by magnitude if
%
%          |d_ii| <= reltol * max(m,n) * |d_11|, i = r+1:n,
%
%       then rank is r, and |d_ii|, i = r+1:n, are set to zero.
%
%       normtol is a tolerance that decides if a submatrix, A_s of
%       dimension m-by-n, of A is considered zero. The matrix is
%       set to zero if
%
%          norm(A_s,'fro') <= normtol *  max(m,n) * norm(A,'fro').
%
%       If tolerances are omitted, they default to eps/2.
%
%       Calls lqdqtlt.m, distributed with this routine, and
%       cholp.m and cod.m from Nicholas Higham's Matrix Computation
%       Toolbox, available at:
%       http://www.ma.man.ac.uk/~higham/mctoolbox

if ~isequal(A,A') | ~isequal(B,B')
  error('A and B must be square and symmetric')
end

if length(A) ~= length(B)
  error('A and B must be the same size')
end

if nargin < 3 | isempty(rrd1), rrd1 = 'spec'; end
if nargin < 4 | isempty(rrd3), rrd3 = 'spec'; end
if nargin < 5 | isempty(rrd4), rrd4 = 'svd'; end
if nargin < 6 | isempty(reltol), reltol = eps/2; end
if nargin < 7 | isempty(normtol), normtol = eps/2; end

n = length(A);

% ********** STEP ONE **********
```

```
tol = reltol * n;

if rrd1 == 'chol'
  [R,P,r1] = cholp(B,tol);
  R(r1+1:n,r1+1:n) = eye(n-r1);

  X1inv = R'\P';

  D11inv = 1;

elseif rrd1 == 'spec'
  [Q,D] = eig(B);
  d = diag(D);

  % sort D and Q, descending in value
  [res,i] = sort(-d);
  d = d(i);
  r1 = sum(d >= tol*d(1));
  Q = Q(:,i);

  X1inv = Q';

  D11inv = diag(1./(sqrt(d(1:r1))));

else
  error('Invalid option for RRD1')

end

% Transform A
A = X1inv*A*X1inv';

% ensure symmetry
A = (A+A')/2;

% ********** STEP TWO **********

A(1:r1,1:r1) = D11inv*A(1:r1,1:r1)*D11inv;

% if r2 = 0 we can solve, else
r2 = n - r1;
if r2 ~= 0
```

```
A(1:r1,r1+1:n) = D11inv*A(1:r1,r1+1:n);

% ********** STEP THREE **********

tol = reltol * r2;
if rrd3 == 'spec'
  [X3invT,D] = eig(A(r1+1:n,r1+1:n));

  % sort D and X3invT, descending in abs value
  d = diag(D);
  [res,i] = sort(-abs(d));
  d = d(i);
  X3invT = X3invT(:,i);

  r3 = sum(abs(d) >= tol*abs(d(1)));

  D=diag(d);

elseif rrd3 == 'ldlt'
  [Q,L,D,P] = lqdqtlt(A(r1+1:n,r1+1:n));

  % sort D and Q, descending in abs value
  d=diag(D);
  [res,i] = sort(-abs(d));
  d = d(i);
  Q = Q(:,i);

  r3 = sum(abs(d) >= tol*abs(d(1)));

  D = diag(d);

  X3invT = P'/L'*Q;

else
  error('Invalid option for RRD3')

end

A(1:r1,r1+1:n) = A(1:r1,r1+1:n)*X3invT;
A(r1+1:r1+r3,r1+1:r1+r3) = D(1:r3,1:r3);

r4 = r2-r3;
```

```
% A13 may be considered zero, accumulate norm(A, 'fro') and test
normA = norm(A(1:r1,1:r1),'fro')^2;
normA = normA+2*norm(A(1:r1,r1+1:n),'fro')^2+sum(diag(D).^2);
normA = sqrt(normA);
normA13 = norm(A(1:r1,n-r4+1:n),'fro');
normtol*normA;
% treat A13 as zero
if normA13  <= normtol*max(r1,r4)*normA
  r5 = 0; r6 = r1; r7 = r4;

% A13 exists and is non zero
elseif r4 ~= 0

% ********** STEP FOUR **********

  tol = reltol * max(r1,r4);
  % We do A13 = X4*Delta*Z4
  if rrd4 == 'qrp'
    [X4,R,P] = qr(A(1:r1,n-r4+1:n));

    % qr with p sorts diag
    if size(R,2) == 1 ||  size(R,1) == 1
       d = R;
       r5 = 1;
       Delta14 = d(1);
    else
       d = diag(R);
       r5 = sum(abs(d) >= tol*abs(d(1)));
       Delta14 = diag(d(1:r5));
    end

    U = eye(r4);

    U(1:r5,:) = Delta14\R(1:r5,:);

    Z4inv = P/U;

  elseif rrd4 == 'cod'
    [X4,Delta14,Z4] = cod(A(1:r1,n-r4+1:n),tol);

    r5 = size(Delta14,1);
```

```
    Z4inv = Z4';

  elseif rrd4 == 'svd'
    [X4,D,Z4inv] = svd(A(1:r1,n-r4+1:n));

    if size(D,2) == 1 ||  size(D,1) == 1
       d = D;
    else
       d = diag(D);
    end

    r5 = sum(abs(d) >= tol*abs(d(1)));

    Delta14 = diag(d(1:r5));

  elseif rrd4 == 'qr'
    [X4,R] = qr(A(1:r1,n-r4+1:n));

    d = diag(R);
    [res,i] = sort(-abs(d));
    d=d(i);

    r5 = sum(abs(d) >= tol*abs(d(1)));

    if r5 < r4
      error('Fix & Heiberger failure - pencil nonregular')
    end

    Delta14 = triu(R(1:r4,:));
    Z4inv = eye(r4);

  else
    error('Invalid option for RRD4')

  end

  r6 = r1-r5;
  r7 = r4-r5;

  A(1:r1,1:r1) = X4'*A(1:r1,1:r1)*X4;
  A(1:r1,r1+1:r1+r3) = X4'*A(1:r1,r1+1:r1+r3);

else
```

```
    r5 = 0; r6 = r1; r7 = 0;

  end

else

    r3 = 0; r4=0; r5 = 0; r6 = r1; r7 = 0;

end

% ********** Solve SEP **********

% B had full rank
if r1 == n

  [U,D] = eig(A);

  U = D11inv*U;
  U = X1inv'*U;

else

  % A22 had full rank in STEP 2 or A13 zero in STEP 3
  if r4 == 0 || r5 == 0

    A11 = A(1:r1,1:r1);
    % generic if r3 = n-r1 last index is just n as before
    A12 = A(1:r1,r1+1:r1+r3);
    D22inv = diag(1./(diag(A(r1+1:r1+r3,r1+1:r1+r3))));

    % if r4 = 0 then r3 = r2 so ok
    U = zeros(n,r1+r3);
    [U(1:r1,1:r1),D] = eig(A11-A12*D22inv*A12');

    U(r1+1:r1+r3,:) = -D22inv*A12'*U(1:r1,:);

    % Set part of U for infinite eigenvalues
    U(r1+1:r1+r3,r1+1:r1+r3) = eye(r3);

    U(r1+1:n,:) = X3invT*U(r1+1:n,:);
    U(1:r1,:) = D11inv*U(1:r1,:);
    U = X1inv'*U;
```

```
  % A14 exists in STEP 4
  else

    A22 = A(r5+1:r1,r5+1:r1);
    A23 = A(r5+1:r1,r1+1:r1+r3);
    D33inv = diag(1./(diag(A(r1+1:r1+r3,r1+1:r1+r3))));
    A12 = A(1:r5,r5+1:r1);
    A13 = A(1:r5,r1+1:r1+r3);

    U = zeros(n,r6+r3+r5);

    [U(r5+1:r1,1:r6),D] = eig(A22-A23*D33inv*A23');

    U(r1+1:r1+r3,:) = -D33inv*A23'*U(r5+1:r1,:);
    U(r1+r3+1:r1+r3+r5,:) = -Delta14\(A12*U(r5+1:r1,:) + ...
                                          A13*U(r1+1:r1+r3,:));

    % Set part of U for infinite eigenvalues
    U(r1+1:r1+r3+r5,r6+1:r6+r3+r5) = eye(r3+r5);

    U(1:r1,:) = X4*U(1:r1,:);
    U(r1+r3+1:n,:) = Z4inv*U(r1+r3+1:n,:);
    U(r1+1:n,:) = X3invT*U(r1+1:n,:);
    U(1:r1,:) = D11inv*U(1:r1,:);
    U = X1inv'*U;

  end

end

r=[r1 r2 r3 r4 r5 r6 r7];
```

C.2 lqdqtlt.m

```
function [Q,L,D,P]=lqdqtlt(A, piv)
%LQDQTLT  LQDQ^TL^T factorization for a symmetric indefinite matrix.
%         Given a Hermitian matrix A,
%         [Q,L,D,P] = lqdqtlt(A, PIV) computes a permutation P,
%         a unit lower triangular L, a real diagonal D and orthogonal
%         Q with 1x1 and 2x2 diagonal blocks, such that
%            P*A*P' = L*Q*D*Q'L'.
%         LDLT_SYMM is called and the resulting 2x2 blocks in D
%         are diagonalized with a spectral decomposition
%
%         PIV controls the pivoting strategy for the LDL^T
%         factorization performed by LDLT_SYMM:
%             PIV = 'p': partial pivoting (Bunch and Kaufman),
%             PIV = 'r': rook pivoting (Ashcroft, Grimes and Lewis).
%         The default is rook pivoting.
%
%         Calls ldlt_symm.m

n = size(A,1);

if nargin == 1
   piv = 'r';
end

% ldlt_symm tests for symmetric A
[L,D,P,rho,ncomp] = ldlt_symm(A, piv);

Q = eye(n);

for j = 1:n-1

   if D(j+1,j) ~= 0
      [Q2,D2] = eig(D(j:j+1,j:j+1));
      D(j:j+1,j:j+1) = D2;
      Q(j:j+1,j:j+1) = Q2;
      j = j+1;
   end

end
```

C.3 gen_data.m

```
function [A,B]=gen_data(r1,r2,r3,r5,k2one,k2two,reg)
%GEN_DATA  Generation of random matrix pencils.
%          [A,B]=gen_data(r1,r2,r3,r5) generates the matrix pencils
%          described below.
%
%                   r1  r2                  r1   r2
%          B = Q' [ D   0 ] Q r1,  A = [ A11  A12 ] r1 , such that
%                 [ 0   0 ]   r2       [ A12' A22 ] r2
%
%          if reg == 1
%          D = diag(lambda_1, ..., lambda_r1)
%          where lambda_i > 0 and geometrically decreases such that
%          lambda_1 = 1 and lambda_r1 = k2one^{-1}
%          and Q is a random orthogonal matrix
%
%          else
%          D = eye(r1) and Q = eye(n)
%
%          A11 is random symmetric matrix
%
%          if reg == 1
%          A22 = Q * diag(lambda_1, ..., lambda_r2) * Q'
%          where lambda_i are geometrically decreasing such that
%          lambda_1 = 1 and lambda_r2 = k2two{-1}
%          and Q is a random orthogonal matrix.
%
%          else
%          A22 = Q * diag(lambda_1, ..., lambda_r3, 0, ..., 0) * Q'
%          where lambda_i are geometrically decreasing such that
%          lambda_1 = 1 and lambda_r3 = k2one{-1}
%
%          if reg == 1
%          A12 is a random matrix
%
%          else
%          A12 =  Q1 * diag(sigma_1, ..., sigma_r5, 0, ..., 0) * Q2
%          where sigma_i are geometrically decreasing such that
%          sigma_1 = 1 and sigma_r5 = k2two{-1}
%          and Q1 and Q2 are random orthogonal matrices.
%
%          if reg == 1 r3 and r5 are not used but are required
```

```
%
%          calls qmult.m (private to elmat.m)

n = r1+r2;
if reg == 1

   % Set B with rank r1
   B = zeros(n);
   d = zeros(r1,1);
   d(1) = 1;

   beta = power(1/k2one,1/(r1-1));
   for m = 2:r1
      d(m) = beta^(m-1);
   end

   B(1:r1,1:r1) = diag(d);
   Q = qmult(n);
   B = Q'*B*Q;
   B = (B+B')/2;

   % Set A11 as random symmetric matrix
   A = zeros(n);

   A(1:r1,1:r1) = rand(r1);
   A(1:r1,1:r1) = (A(1:r1,1:r1)+A(1:r1,1:r1)')/2;

   % Set A22 with full rank
   d = zeros(r2,1);
   d(1) = 1;

   beta = power(1/k2two,1/(r2-1));
   for m = 2:r2
      d(m) = beta^(m-1);
   end

   A(r1+1:n,r1+1:n) = diag(d);
   Q = qmult(r2);
   A(r1+1:n,r1+1:n) = Q*A(r1+1:n,r1+1:n)*Q';
   A(r1+1:n,r1+1:n) = (A(r1+1:n,r1+1:n)+A(r1+1:n,r1+1:n)')/2;

   % Set A12 as random matrix
   A(1:r1,r1+1:n) = rand(r1,r2);
```

```
   A(r1+1:n,1:r1) = A(1:r1,r1+1:n)';

else

   % Set B with rank r1
   B = zeros(n);
   B(1:r1,1:r1) = eye(r1);

   % Set A11 as random symmetric matrix
   A = zeros(n);

   A(1:r1,1:r1) = rand(r1);
   A(1:r1,1:r1) = (A(1:r1,1:r1)+A(1:r1,1:r1)')/2;

   % Set A22 with rank r3
   d(1) = 1;

   beta = power(1/k2one,1/(r3-1));
   for m = 2:r3
      d(m) = beta^(m-1);
   end

   A(r1+1:r1+r3,r1+1:r1+r3) = diag(d);
   Q = qmult(r2);
   A(r1+1:n,r1+1:n) = Q*A(r1+1:n,r1+1:n)*Q';
   A(r1+1:n,r1+1:n) = (A(r1+1:n,r1+1:n)+A(r1+1:n,r1+1:n)')/2;

   % Set A12 with rank r5
   d = zeros(r5,1);
   d(1) = 1;

   beta = power(1/k2two,1/(r5-1));
   for m = 2:r5
      d(m) = beta^(m-1);
   end

   A(1:r5,r1+1:r1+r5) = diag(d);
   Q2 = qmult(r2);
   Q1 = qmult(r1);
   A(1:r1,r1+1:n) = Q1*A(1:r1,r1+1:n)*Q2;
   A(r1+1:n,1:r1) = A(1:r1,r1+1:n)';

end
```

Appendix D

Code for Updating the QR Factorization

D.1 delcols.f

```
      SUBROUTINE DELCOLS( M, N, A, LDA, K, P, TAU, WORK, INFO )
*
*     Craig Lucas, University of Manchester
*     March, 2004
*
*     .. Scalar Arguments ..
      INTEGER            INFO, K, LDA, M, N, P
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), TAU( * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  Given a real m by (n+p) matrix, B, and the QR factorization
*  B = Q_B * R_B, DELCOLS computes the QR factorization
*  C = Q * R where C is the matrix B with p columns deleted
*  from the kth column onwards.
*
*  The input to this routine is Q_B' * C
*
*  Arguments
*  =========
*
```

```
*  M       (input) INTEGER
*          The number of rows of the matrix C.  M >= 0.
*
*  N       (input) INTEGER
*          The number of columns of the matrix C.  N >= 0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the matrix Q_B' * C. The elements in columns
*          1:K-1 are not referenced.
*
*          On exit, the elements on and above the diagonal contain
*          the n by n upper triangular part of the matrix R. The
*          elements below the diagonal in columns k:n, together with
*          TAU represent the orthogonal matrix Q as a product of
*          elementary reflectors (see Further Details).
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,M).
*
*  K       (input) INTEGER
*          The position of the first column deleted from B.
*          0 < K <= N+P.
*
*  P       (input) INTEGER
*          The number of columns deleted from B.  P > 0.
*
*  TAU     (output) DOUBLE PRECISION array, dimension(N-K+1)
*          The scalar factors of the elementary reflectors
*          (see Further Details).
*
*  WORK    DOUBLE PRECISION array, dimension (P+1)
*          Work space.
*
*  INFO    (output) INTEGER
*          = 0: successful exit
*          < 0: if INFO = -I, the I-th argument had an illegal value.
*
*  Further Details
*  ===============
*
*  The matrix Q is represented as a product of Q_B and elementary
*  reflectors
*
```

```
*     Q = Q_B * H(k) * H(k+1) *...* H(last), last = min( m-1, n ).
*
*  Each H(j) has the form
*
*     H(j) = I - tau*v*v'
*
*  where tau is a real scalar, and v is a real vector with
*  v(1:j-1) = 0, v(j) = 1, v(j+1:j+lenh-1), lenh = min( p+1, m-j+1 ),
*  stored on exit in A(j+1:j+lenh-1,j) and v(j+lenh:m) = 0, tau is
*  stored in  TAU(j).
*
*  The matrix Q can be formed with DELCOLSQ
*
*  ====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE
      PARAMETER          ( ONE = 1.0D+0 )
*     ..
*     .. Local Scalars ..
      DOUBLE PRECISION   AJJ
      INTEGER            J, LAST, LENH
*     ..
*     .. External Subroutines ..
      EXTERNAL           DLARF, DLARFG, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX, MIN
*     ..
*
*     Test the input parameters.
*
      INFO = 0
      IF( M.LT.0 ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, M ) ) THEN
         INFO = -4
      ELSE IF( K.GT.N+P .OR. K.LE.0 ) THEN
         INFO = -5
      ELSE IF( P.LE.0 ) THEN
         INFO = -6
```

```
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'DELCOLS', -INFO )
         RETURN
      END IF
*
      LAST = MIN( M-1, N )
*
      DO 10 J = K, LAST
*
*        Generate elementary reflector H(J) to annihilate the nonzero
*        entries below A(J,J)
*
         LENH = MIN( P+1, M-J+1 )
         CALL DLARFG( LENH, A( J, J ), A( J+1, J ), 1, TAU( J-K+1 ) )
*
         IF( J.LT.N ) THEN
*
*           Apply H(J) to trailing matrix from left
*
            AJJ = A( J, J )
            A( J, J ) = ONE
            CALL DLARF( 'L', LENH, N-J, A( J, J ), 1, TAU( J-K+1 ),
     $                  A( J, J+1 ), LDA, WORK )
            A( J, J ) = AJJ
*
         END IF
*
   10 CONTINUE
*
      RETURN
*
*     End of DELCOLS
*
      END
```

D.2 delcolsq.f

```
      SUBROUTINE DELCOLSQ( M, N, A, LDA, Q, LDQ, K, P, TAU, WORK, INFO )
*
*     Craig Lucas, University of Manchester
*     March, 2004
*
*     .. Scalar Arguments ..
      INTEGER            INFO, K, LDA, LDQ, M, N, P
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), Q( LDQ, * ), TAU( * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  DELCOLSQ generates an m by m real matrix Q with orthogonal columns,
*  which is defined as the product of Q_B and elementary reflectors
*
*        Q = Q_B * H(k) * H(k+1) *...* H(last), last = min( m-1, n ) .
*
*  where the H(j) are as returned by DELCOLSQ, such that C = Q * R and
*  C is the matrix B = Q_B * R_B, with p columns deleted from the
*  kth column onwards.
*
*  Arguments
*  =========
*
*  M       (input) INTEGER
*          The number of rows of the matrix A.  M >= 0.
*
*  N       (input) INTEGER
*          The number of columns of the matrix A.  N >= 0.
*
*  A       (input) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the elements below the diagonal in columns k:n
*          must contain the vector which defines the elementary
*          reflector H(J) as returned by DELCOLS.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,M).
*
```

```
*  Q       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the matrix Q_B.
*          On exit, the matrix Q.
*
*  LDQ     (input) INTEGER
*          The leading dimension of the array Q.  LDQ >= M.
*
*  K       (input) INTEGER
*          The position of the first column deleted from B.
*          0 < K <= N+P.
*
*  P       (input) INTEGER
*          The number of columns deleted from B.  P > 0.
*
*  TAU     (input) DOUBLE PRECISION array, dimension(N-K+1)
*          TAU(J) must contain the scalar factor of the elementary
*          reflector H(J), as returned by DELCOLS.
*
*  WORK    DOUBLE PRECISION array, dimension (P+1)
*          Work space.
*
*  INFO    (output) INTEGER
*          = 0: successful exit
*          < 0: if INFO = -I, the I-th argument had an illegal value.
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE
      PARAMETER          ( ONE = 1.0D+0 )
*     ..
*     .. Local Scalars ..
      DOUBLE PRECISION   AJJ
      INTEGER            J, LAST, LENH
*     ..
*     .. External Subroutines ..
      EXTERNAL           DLARF, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX, MIN
*     ..
*
*     Test the input parameters.
```

```
*
      INFO = 0
      IF( M.LT.0 ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, M ) ) THEN
         INFO = -4
      ELSE IF( K.GT.N+P .OR. K.LE.0 ) THEN
         INFO = -5
      ELSE IF( P.LE.0 ) THEN
         INFO = -6
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'DELCOLSQ', -INFO )
         RETURN
      END IF
*
      LAST = MIN( M-1, N )
*
      DO 10 J = K, LAST
*
         LENH = MIN( P+1, M-J+1 )
*
*        Apply H(J) from right
*
         AJJ = A( J, J )
         A( J, J ) = ONE
*
         CALL DLARF( 'R', M, LENH, A( J, J ), 1, TAU( J-K+1 ),
     $               Q( 1, J ), LDQ, WORK )
*
         A( J, J ) = AJJ
*
   10 CONTINUE
*
      RETURN
*
*     End of DELCOLSQ
*
      END
```

D.3 addcols.f

```
      SUBROUTINE ADDCOLS( M, N, A, LDA, K, P, TAU, WORK, LWORK, INFO )
*
*     Craig Lucas, University of Manchester
*     March, 2004
*
*     .. Scalar Arguments ..
      INTEGER            INFO, K, LDA, LWORK, M, N, P
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), TAU( * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  Given a real m by (n-p) matrix, B, and the QR factorization
*  B = Q_B * R_B, ADDCOLS computes the QR factorization
*  C = Q * R where C is the matrix B with p columns added
*  in the kth column onwards.
*
*  The input to this routine is Q_B' * C
*
*  Arguments
*  =========
*
*  M       (input) INTEGER
*          The number of rows of the matrix C.  M >= 0.
*
*  N       (input) INTEGER
*          The number of columns of the matrix C.  N >= 0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the matrix Q_B' * C. The elements in columns
*          1:K-1 are not referenced.
*
*          On exit, the elements on and above the diagonal contain
*          the n by n upper triangular part of the matrix R. The
*          elements below the diagonal in columns K:N, together with
*          TAU represent the orthogonal matrix Q as a product of
*          elementary reflectors and Givens rotations.
*          (see Further Details).
```

```
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,M).
*
*  K       (input) INTEGER
*          The position of the first column added to B.
*          0 < K <= N-P+1.
*
*  P       (input) INTEGER
*          The number of columns added to B.  P > 0.
*
*  TAU     (output) DOUBLE PRECISION array, dimension(P)
*          The scalar factors of the elementary reflectors
*          (see Further Details).
*
*  WORK    (workspace) DOUBLE PRECISION array, dimension ( LWORK )
*          Work space.
*
*  LWORK   (input) INTEGER
*          The dimension of the array WORK.  LWORK >= P.
*          For optimal performance LWORK >= P*NB, where NB is the
*          optimal block size.
*
*  INFO    (output) INTEGER
*          = 0: successful exit
*          < 0: if INFO = -I, the I-th argument had an illegal value.
*
*  Further Details
*  ===============
*
*  The matrix Q is represented as a product of Q_B, elementary
*  reflectors and Givens rotations
*
*     Q = Q_B * H(k) * H(k+1) *...* H(k+p-1) * G(k+p-1,k+p) *...
*         *G(k,k+1) * G(k+p,k+p+1) *...* G(k+2p-2,k+2p-1)
*
*  Each H(j) has the form
*
*     H(j) = I - tau*v*v'
*
*  where tau is a real scalar, and v is a real vector with
*  v(1:n-p-j+1) = 0, v(j) = 1, and v(j+1:m) stored on exit in
*  A(j+1:m,j), tau is stored in TAU(j).
```

```
*
*  Each G(i,j) has the form
*
*                  i-1  i
*              [ I          ]
*              [   c   -s   ] i-1
*     G(i,j) = [   s    c   ] i
*              [          I ]
*
*  and zero A(i,j), where c and s are encoded in scalar and
*  stored in A(i,j) and
*
*     IF A(i,j) = 1, c = 0, s = 1
*     ELSE IF | A(i,j) | < 1, s = A(i,j), c = sqrt(1-s**2)
*     ELSE c = 1 / A(i,j), s = sqrt(1-c**2)
*
*  The matrix Q can be formed with ADDCOLSQ
*
*  ====================================================================
*
*     .. Local Scalars ..
      DOUBLE PRECISION   C, S
      INTEGER            I, INC, ISTART, J, JSTOP, UPLEN
*     ..
*     .. External Subroutines ..
      EXTERNAL           DGEQRF, DLASR, DROT, DROTG, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX, MIN
*     ..
*
*     Test the input parameters.
*
      INFO = 0
      IF( M.LT.0 ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, M ) ) THEN
         INFO = -4
      ELSE IF( K.GT.N-P+1 .OR. K.LE.0 ) THEN
         INFO = -5
      ELSE IF( P.LE.0 ) THEN
```

```
         INFO = -6
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'ADDCOLS', -INFO )
         RETURN
      END IF
*
*     Do a QR factorization on rows below N-P, if there is more than one
*
      IF( M.GT.N-P+1 ) THEN
*
*        Level 3 QR factorization
*
         CALL DGEQRF( M-N+P, P, A( N-P+1, K ), LDA, TAU, WORK, LWORK,
     $                INFO )
*
      END IF
*
*     If K not equal to number of columns in B and not <= M-1 then
*     there is some elimination by Givens to do
*
      IF( K+P-1.NE.N .AND. K.LE.M-1) THEN
*
*        Zero out the rest with Givens
*        Allow for M < N
*
         JSTOP = MIN( P+K-1, M-1 )
         DO 20 J = K, JSTOP
*
*           Allow for M < N
*
            ISTART = MIN( N-P+J-K+1, M )
            UPLEN = N - K - P - ISTART + J + 1
*
            INC = ISTART - J
*
            DO 10 I = ISTART, J + 1, -1
*
*              Recall DROTG updates A( I-1, J ) and
*              stores C and S encoded as scalar in A( I, J )
*
               CALL DROTG( A( I-1, J ), A( I, J ), C, S )
               WORK( INC ) = C
```

```
               WORK( N+INC ) = S
*
*              Update nonzero rows of R
*              Do the next two line this way round because
*              A( I-1, N-UPLEN+1 ) gets updated
*
               A( I, N-UPLEN ) = -S*A( I-1, N-UPLEN )
               A( I-1, N-UPLEN ) = C*A( I-1, N-UPLEN )
*
               CALL DROT( UPLEN, A( I-1, N-UPLEN+1 ), LDA,
     $                    A( I, N-UPLEN+1 ), LDA, C, S )
*
               UPLEN = UPLEN + 1
               INC = INC - 1
*
   10       CONTINUE
*
*           Update inserted columns in one go
*           Max number of rotations is N-1, we've allowed N
*
            IF( J.LT.P+K-1 ) THEN
*
               CALL DLASR( 'L', 'V', 'B', ISTART-J+1, K+P-1-J,
     $                     WORK( 1 ), WORK( N+1 ), A( J, J+1 ), LDA )
*
            END IF
*
   20    CONTINUE
*
      END IF
      RETURN
*
*     End of ADDCOLS
*
      END
```

D.4 addcolsq.f

```
      SUBROUTINE ADDCOLSQ( M, N, A, LDA, Q, LDQ, K, P, TAU, WORK, INFO)
*
*     Craig Lucas, University of Manchester
*     March, 2004
*
*     .. Scalar Arguments ..
      INTEGER            INFO, K, LDA, LDQ, M, N, P
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), Q( LDQ, * ), TAU( * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  ADDCOLSQ generates an m by m real matrix Q with orthogonal columns,
*  which is defined as the product of Q_B, elementary  reflectors and
*  Givens rotations
*
*     Q = Q_B * H(k) * H(k+1) *...* H(k+p-1) * G(k+p-1,k+p) *...
*         *G(k,k+1) * G(k+p,k+p+1) *...* G(k+2p-2,k+2p-1)
*
*  where the H(j) and G(i,j) are as returned by ADDCOLS, such that
*  C = Q * R and C is the matrix B = Q_B * R_B, with p columns added
*  from the kth column onwards.
*
*  Arguments
*  =========
*
*  M       (input) INTEGER
*          The number of rows of the matrix A.  M >= 0.
*
*  N       (input) INTEGER
*          The number of columns of the matrix A.  N >= 0.
*
*  A       (input) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the elements below the diagonal in columns
*          K:K+P-1 (if M > M-P+1) must contain the vector which defines
*          the elementary reflector H(J). The elements above these
*          vectors and below the diagonal store the scalars such that
*          the Givens rotations can be constructed, as returned by
```

```
*          ADDCOLS.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,M).
*
*  Q       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the matrix Q_B.
*          On exit, the matrix Q.
*
*  LDQ     (input) INTEGER
*          The leading dimension of the array Q.  LDQ >= M.
*
*  K       (input) INTEGER
*          The postion of first column added to B.
*          0 < K <= N-P+1.
*
*  P       (input) INTEGER
*          The number columns added.  P > 0.
*
*  TAU     (output) DOUBLE PRECISION array, dimension(N-K+1)
*          The scalar factors of the elementary reflectors.
*
*  WORK    (workspace) DOUBLE PRECISION array, dimension (2*N)
*          Work space.
*
*  INFO    (output) INTEGER
*          = 0: successful exit
*          < 0: if INFO = -I, the I-th argument had an illegal value
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER          ( ONE = 1.0D+0, ZERO = 0.0D+0 )
*     ..
*     .. Local Scalars ..
      DOUBLE PRECISION   DTEMP
      INTEGER            COL, I, INC, ISTART, J, JSTOP
*     ..
*     .. External Subroutines ..
      EXTERNAL           DLARF, DLASR, XERBLA
*     ..
*     .. Intrinsic Functions ..
```

```
      INTRINSIC          ABS, MAX, MIN, SQRT
*
*     Test the input parameters.
*
      INFO = 0
      IF( M.LT.0 ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, M ) ) THEN
         INFO = -4
      ELSE IF( K.GT.N-P+1 .OR. K.LE.0 ) THEN
         INFO = -5
      ELSE IF( P.LE.0 ) THEN
         INFO = -6
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'ADDCLQ', -INFO )
         RETURN
      END IF
*
*     We did a QR factorization on rows below N-P+1
*
      IF( M.GT.N-P+1 ) THEN
*
         COL = N - P + 1
         DO 10 J = K, K + P - 1
*
            DTEMP = A( COL, J )
            A( COL, J ) = ONE
*
*           If  N+P > M-N we have only factored the first M-N columns.
*
            IF( M-COL+1.LE.0 )
     $         GO TO 10
            CALL DLARF( 'R', M, M-COL+1, A( COL, J ), 1, TAU( J-K+1 ),
     $                  Q( 1, COL ), LDQ, WORK )
*
            A( COL, J ) = DTEMP
            COL = COL + 1
*
   10    CONTINUE
      END IF
```

```
*
*     If K not equal to number of columns in B then there was
*     some elimination by Givens
*
      IF( K+P-1.LT.N .AND. K.LE.M-1 ) THEN
*
*        Allow for M < N, i.e DO P wide unless hit the bottom first
*
         JSTOP = MIN( P+K-1, M-1 )
         DO 30 J = K, JSTOP
*
            ISTART = MIN( N-P+J-K+1, M )
            INC = ISTART - J
*
*           Compute vectors of C and S for rotations
*
            DO 20 I = ISTART, J + 1, -1
*
               IF( A( I, J ).EQ.ONE ) THEN
                  WORK( INC ) = ZERO
                  WORK( N+INC ) = ONE
               ELSE IF( ABS( A( I, J ) ).LT.ONE ) THEN
                  WORK( N+INC ) = A( I, J )
                  WORK( INC ) = SQRT( ( 1-A( I, J )**2 ) )
               ELSE
                  WORK( INC ) = ONE / A( I, J )
                  WORK( N+INC ) = SQRT( ( 1-WORK( INC )**2 ) )
               END IF
               INC = INC - 1
   20       CONTINUE
*
*           Apply rotations to the Jth column from the right
*
            CALL DLASR( 'R', 'V', 'b', M, ISTART-I+1, WORK( 1 ),
     $                  WORK( N+1 ), Q( 1, I ), LDQ )
*
   30    CONTINUE
*
      END IF
      RETURN
*
*     End of ADDCOLS
*
```

```
END
```

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Third edition, SIAM, Philadelphia, 1999.

[2] Cleve Ashcraft, Roger G. Grimes, and John G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.*, 20(2):513–561, 1998.

[3] Automatically Tuned Linear Algebra Software (ATLAS). `http://math-atlas.sourceforge.net/ /`.

[4] Å. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, PA, USA, 1996.

[5] Å. Björck, L. Eldén, and H. Park. Accurate downdating of least squares solutions. *SIAM J. Matrix Anal. Appl.*, 15:549–568, 1994.

[6] A. W. Bojanczyk, R. P. Brent, P. Van Dooren, and F. R. De Hoog. A note on downdating the Cholesky factorization. *SIAM J. Sci. Stat. Comput.*, 8(3):210–221, 1987.

[7] Adam Bojanczyk, Nicholas J. Higham, and Harikrishna Patel. Solving the indefinite least squares problem by hyperbolic QR factorization. *SIAM J. Matrix Anal. Appl.*, 24(4):914–931, 2003.

[8] James R. Bunch and Linda Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31(137):163–179, 1977.

[9] A. Bunse-Gerstner. An algorithm for the symmetric generalized eigenvalue problem. *Linear Algebra and Appl.*, 58:43–68, 1984.

[10] Ralph Byers, Volker Mehrmann, and Hongguo Xu. A structured staircase algorithm for skew-symmetric/symmetric pencils. In preparation.

[11] Zhi Hao Cao. On a deflation method for the symmetric generalized eigenvalue problem. *Linear Algebra and Appl.*, 92:187–196, 1987.

[12] J. M. Chambers. Regression updating. *Journal of the American Statistical Association*, 66(336):744–748, 1971.

[13] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the Gram–Schmidt QR factorization. *Mathematics of Computation*, 30(136):772–795, 1976.

[14] J. W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[15] James Demmel and Bo Kågström. The generalized Schur decomposition of an arbitrary pencil $A - \lambda B$: Robust software with error bounds and applications. Part I: Theory and algorithms. *ACM Trans. Math. Soft.*, 19 (2):160–174, 1993.

[16] James Demmel and Bo Kågström. The generalized Schur decomposition of an arbitrary pencil $A - \lambda B$: Robust software with error bounds and applications. Part II: Software and applications. *ACM Trans. Math. Soft.*, 19(2):175–201, 1993.

[17] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Prog.*, 91:201–213, 2002.

[18] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, 1990.

[19] J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenproblem. *SIAM J. Sci. Stat. Comput.*, 8(2):139–154, 1987.

[20] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Soft.*, 14(1):18–32, 1988.

[21] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979.

[22] L. Eldén and H. Park. Block downdating of least squares solutions. *SIAM J. Matrix Anal. Appl.*, 15:1018–1034, 1994.

[23] G. Fix and R. Heiberger. An algorithm for the ill-conditioned generalized eigenvalue problem. *SIAM Journal on Numerical Analysis*, 9(1):78–88, 1972.

[24] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Third edition, The Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[25] D. J. Higham and N. J. Higham. Structured backward error and condition of generalized eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 20(2): 493–512, 1998.

[26] Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005. To be published.

[27] N. J. Higham. Analysis of the Cholesky decomposition of a semidefinite matrix. In *Reliable Numerical Computation*, M. G. Cox and S. J. Hammarling, editors, Oxford University Press, 1990, pages 161–185.

[28] Nicholas J. Higham. The Matrix Computation Toolbox. `http://www.ma.man.ac.uk/~higham/mctoolbox`.

[29] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[30] R. A. Horn and C. R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

[31] B. Kågström. Singular matrix pencils (section 8.7). In *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*, Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors, Society for Industrial and Applied Mathematics, 2000, pages 260–277.

[32] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3): 308–323, 1979.

[33] Craig Lucas. *Computing Nearest Covariance and Correlation Matrices*. MSc thesis, University of Manchester, Manchester, England, 2001.

[34] Matrix Market. `http://math.nist.gov/MatrixMarket/`.

[35] `/matlab6.5/toolbox/matlab/elmat/private/qmult.m` from the MATLAB distibution.

[36] MATLAB, version 6.5. The Mathworks Inc, Natick, MA, USA.

[37] C. B. Moler and G. W. Stewart. An algorithm for generalized matrix eigenvalue problems. *SIAM Journal on Numerical Analysis*, 10(2):241–256, 1973.

[38] NAG Fortran Library Manual, Mark 20. Numerical Algorithms Group, Oxford, UK.

[39] S. J. Olszanskyj, J. M. Lebak, and A. W. Bojanczyk. Rank-k modificaton methods for recursive least squares problems. *Numerical Algorithms*, 7: 325–354, 1994.

[40] B. N. Parlett. Analysis of algorithms for reflections in bisectors. *SIAM Review*, 13:197–208, 1971.

[41] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[42] C. M. Rader and A. O. Steinhardt. Hyperbolic Householder transforms. *SIAM J. Matrix Anal. Appl.*, 9(2):269–290, 1988.

[43] L. Reichel and W. B. Gragg. Algorithm 686: FORTRAN subroutines for updating the QR decomposition. *ACM Trans. Math. Soft.*, 16:369–377, 1990.

[44] M. A. Saunders. Large-scale linear programming using the cholesky factorization. Technical Report CS252, Computer Science Department, Stanford University, CA, 1972.

[45] Robert Schreiber and Charles van Loan. A storage-efficient wy representation for products of householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, 1989.

[46] BLAS Technical Forum Standard. `http://www.netlib.org/blas/blast-forum/`.

[47] G. W. Stewart. The effects of rounding error on an algorithm for downdating a Cholesky factorization. *Journal of the Institute of Mathematics and its Applications*, 23(2):203–213, 1979.

[48] G. W. Stewart. The efficient generation of random orthogonal matrices with an application to condition estimators. *SIAM Journal on Numerical Analysis*, 17(3):403–409, 1980.

[49] G. W. Stewart. On the stability of sequential updates and downdates. *IEEE Trans. Signal Processing*, 43(11):2642–2648, 1995.

[50] G. W. Stewart. *Matrix Algorithms. Volume II: Eigensystems.* SIAM, Philadelphia, PA, USA, 2001.

[51] L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, Philadelphia, PA, USA, 1997.