# Chapter 13 Software Documentation

Most of the software used for the examples in this book is available online at ftp://ftp.wiley.com/public/sci_tech_med/least_squares/. It is the author's intent to continue enhancing and augmenting the code, and to describe status and updates in a README file. Please consult the README file for the current status.

Before describing the software, we define the terms of use as described in the prolog of each code module. The software is licensed under the following terms:

All source code is written in Fortran 90/95 and has been tested using the Intel Fortran 9.1.033 and Compaq Visual Fortran 6.6 compilers. These compilers were intentionally chosen to ensure that the code did not depend on recent compiler enhancements. Limited testing was also conducted using the Intel Fortran 11.1.065 (March 2010) compiler. Special features of the compilers have not been used, so the code should compile and execute correctly on most conforming Fortran 95 or later compilers. All utility routines or modules are located in the *Library* directory, with source code for most of the examples in previous chapters located in the corresponding "Chapter" directories and "example name" subdirectories. The example directories also include sample program output.

Most of the associated library software has been used successfully by this author for many years. However, many changes were made in order to create a software package suitable for distribution. Most changes simply involved additions to or modifications of comments, but some required modifications of executable code. The author has attempted to validate these modifications by comparison with the original code or with independent calculations. However, it is not feasible to test all possible variations on code usage, so it cannot be guaranteed that the software will work correctly for your application. As when using any software (yours or from another source), it is always prudent to check the reasonableness of the results and compare with alternate computations.

The software has been coded using the following general guidelines:

1. Standard Fortran 90/95 capabilities have been used extensively to enhance readability of the code, but not to an extent that would make porting of the code to MATLAB or another

language difficult. In particular, Fortran 90/95 vector/matrix operators have been used as much as possible to simplify coding and improve readability.

2. Subroutine calling arguments are ordered as (output, input/output, input) so that the subroutines appear much like functions. This not only enhances code readability, but also makes porting to MATLAB or a simulation language easier. Calling arguments are defined in the code with input/output intent, and trailing comments on the same line describe the variables.
3. In some cases collections of variables, subroutines, and functions were collected into Fortran 90 modules. This encapsulation has a number of benefits, but was not used as much as normally desirable. Limiting use of modules facilitates porting to other languages and simplifies extraction of individual files for incorporation in other programs.
4. Some library functions write text output to Fortran unit 6 (standard output). In many cases this is controlled with a "debug print" calling argument.
5. All programs, subroutines, functions, and modules contain a prolog explaining the purpose and usage of the software, and calling variable definitions. The extent of these prologs is variable as it depends on the heritage of the source code and the available time of this author. Prolog improvements are one priority of future enhancements.
6. Software licensing information is listed in the prolog.

The example directories contain source code for test programs that were used to generate results used in various examples. They generally link other modules from the library directory or from libraries listed above. As with most test programs used to generate output for this book, the code comments are sparse and some options must be selected by recompiling the code. Search for "###" to find variables that may be changed to select different options. Comments near the beginning of each file list the command line for compiling using the Intel Fortran compiler. Also listed, in most cases, is the command line to execute the program.

The following sections describe the modules contained in the corresponding directories. See comments in the code for further information on module usage and calling arguments.

## 13.1 Library

The following sections describe subroutines, functions, and modules included in the library directory. Many routines are used by the example programs located in the chapter directories.

The floating point precision of many filter-related subroutines can be changed from single (4-bytes) to double (8-bytes) precision by changing the *real_kind1* variable in the FILTER_PREC.F module and then recompiling. The subroutines with this feature include COV2UD_M, KF, MATINV_GJ, PHIU_M, PRTMX, PRTSMX, R2A_M, RANK1_M, RDX, RI2COV_M, RINCON_M, RINZ_M, SINV, SMULT, SYMULT, SYSMULT, SRIF, TDHHT_M, THHC_M, TRIPHU, UD2COV_M, UD2SIG_M, UDKF, UDMES_M, and WGS_M. Some example programs also use *real_kind1*.

Several library subroutines used for factorized estimation are enhancements of software originally written by G. Bierman and associates. This code came from two sources. The Fortran 77 Parameter Estimation Subroutine Package (ESP) was created by Bierman and others in 1978 while at NASA/Jet Propulsion Laboratory (Bierman 1977, Bierman and Nead 1978). This software was provided free without restrictions to interested parties. Later Bierman created a commercial version of the estimation subroutines—called the Estimation Subroutine Library

(ESL; Bierman and Bierman 1984)—that is now available at http://netlib.org/a/esl.tgz. These ESL subroutines are licensed under the *Creative Commons* license, so they may be used, copied, and modified according to the license. The modified routines from this source included in this package are licensed under both the *Creative Commons* license and the license listed above.

Several other subroutines are also used in example programs. Because of potential licensing restrictions, these are not directly included in the software package. Sources for these packages are described below. Since web addresses can change, it may be necessary to search for keywords if locations are invalid.

1. LAPACK: Linear algebra package developed by Anderson et al. (1999). This is a collection of Fortran 90 subroutines that have become the standard for linear algebra. See http://www.netlib.org/lapack/ for more information and downloads, or search the web. While it is possible to directly download the source code and then compile all files on a PC, multiple dependencies in the code make operation somewhat erratic unless compilation is done carefully. For use on a PC under Intel Fortran, it is recommended that you download compiled libraries found at http://icl.cs.utk.edu/lapack-for-windows/lapack/index.html#librairies.
2. Bierman's ESL package of Fortran 77 subroutines is used for factorized (U-D or Square Root Information Filter (SRIF)) estimation. See http://netlib.org/a/esl.tgz.
3. The LSQR iterative sparse equation and least-squares solver developed by Paige and Saunders (1982) is used in Chapter 5 examples. More information and downloads may be found at http://www.stanford.edu/group/SOL/software/lsqr.html.
4. The NL2SOL adaptive nonlinear least-squares algorithm was developed by Dennis et al. (1981) and used in Chapter 7 examples. Information and downloads may be found at http://www.netlib.org/toms/573 and http://people.sc.fsu.edu/~burkardt/f_src/nl2sol/nl2sol.html.
5. Secondary example programs (not supplied) in Chapters 4, 5 and 6 use the Numerical Recipes (Press et al. 1992, 2007) subroutines SVDCMP, GAMMAQ and ODEINT. Source code for these subroutines may be downloaded (for a fee) from http://www.nr.com/com/storefront.html. Older versions of the books, including code, can be viewed at http://www.nr.com/oldverswitcher.html.

## 13.1.1 Matrix Operations

The following subroutines and functions are used to perform general matrix operations.

### 13.1.1.1 Subroutine BICGSTAB (ier, xe, A,b,n,nd)

Subroutine BICGSTAB is a preconditioned bi-conjugate gradient stabilized method (see Kelley 1995) for solving $\mathbf{A}\,\mathbf{x}_e = \mathbf{b}$ for $\mathbf{x}_e$ where $\mathbf{A}$ is a $n \times n$. Numerical experience has demonstrated that this method converges reliably on flow-type problems. This version assumes that matrix $\mathbf{A}$ is mostly full. For some problems of this type, $\mathbf{A}$ is very sparse, so it may be desirable to store $\mathbf{A}$ in a sparse format, and modify internal multiplications accordingly.

### 13.1.1.2 Subroutine CFACTOR (ier, A, n,eps)

Subroutine CFACTOR computes the Cholesky factor of a symmetric positive definite matrix $\mathbf{A}$; that is, $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is lower triangular. Matrix $\mathbf{A}$ is stored as upper triangular by columns, and the output $\mathbf{L}^T$ is stored in $\mathbf{A}$. If matrix $\mathbf{A}$ is singular at row *i*, the same row of the output $\mathbf{A}$ will be set to zero and *ier* will be set to the first row that is found to be singular. CFACTOR also tests for a loss of precision and returns an error code indicating at which row the loss occurred.

#### 13.1.1.3 Subroutine CGNR (ier, xe, H,y,n,m,mmax)

Subroutine CGNR is a preconditioned conjugate gradient method for solving the least-squares normal equations to minimize the residual (see Kelley 1995, Björck 1996). The measurement equation is $\mathbf{y} = \mathbf{H}\,\mathbf{x}_e + \mathbf{r}$ with the normal equation $\hat{\mathbf{x}}_e = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{y}$.

#### 13.1.1.4 Function CONDEST (R,n)

Real(8) function CONDEST computes the $l_1$ condition number estimate of an upper triangular matrix $\mathbf{R}$ using the LINPACK algorithm contained in Cline et al. (1979). Also see Algorithm A3.3.1, Dennis and Schnabel (1983).

#### 13.1.1.5 Subroutine FILTERPREC

Subroutine FILTERPREC is a Fortran 90/95/2005 module containing only one variable: integer(4) parameter *real_kind1*. The value of *real_kind1* may be set to 4 for single precision, or 8 for double precision. Module FILTERPREC is referenced in library subroutines COV2UD_M, KF, MATINV_GJ, PHIU_M, PRTMX, PRTSMX, R2A_M, RANK1_M, RDX, RI2COV_M, RINCON_M, RINZ_M, SINV, SMULT, SYMULT, SYSMULT, SRIF, TDHHT_M, THHC_M, TRIPHU, UD2COV_M, UD2SIG_M, UDKF, UDMES_M, and WGS_M. The precision of these subroutines can be set by first compiling FILTERPREC, and then compiling the other routines.

#### 13.1.1.6 Function L1NORMM (C,nd,n)

Real(8) function L1NORMM computes the $l_1$ norm of matrix $\mathbf{C}(n,n)$: $\|\mathbf{C}\|_1 = \max_j \|\mathbf{c}_{:j}\|_1$ where $\mathbf{c}_{:j}$ is the *j*-th column of $\mathbf{C}$.

#### 13.1.1.7 Function L2NORMM (C,nd,n)

Real(8) function L2NORMM computes the Frobenius $l_2$-norm of matrix $\mathbf{C}(n,n)$:

$$\|\mathbf{C}\|_F = \left( \sum_{i=1}^{m} \sum_{j=1}^{n} C_{ij}^2 \right)^{1/2}$$

#### 13.1.1.8 Subroutine MATINV_GJ (ierr, A,B, np,n,m)

Subroutine MATINV_GJ inverts matrix $\mathbf{A}(n,n)$ and computes $\mathbf{A}^{-1}\mathbf{B}$ where matrix $\mathbf{B}$ is dimensioned $\mathbf{B}(n,m)$. On output $\mathbf{A} \Leftarrow \mathbf{A}^{-1}$ and $\mathbf{B} \Leftarrow \mathbf{A}^{-1}\mathbf{B}$. The solution method is Gauss-Jordan elimination with full pivoting. (See Press et al. 2007.)

#### 13.1.1.9 Subroutine PRTMX (A,nn,n,l,ax)

Subroutine PRTMX prints a rectangular matrix, one row at a time, using sparse matrix print logic.

#### 13.1.1.10 Subroutine PRTSMX (A,n,ax)

Subroutine PRTSMX prints the upper triangular portion of a symmetric square matrix (using packed storage), one row at a time, using sparse matrix print logic.

#### 13.1.1.11 Subroutine QRDCMP (Hc,rsos, mmax,n,m)

Subroutine QRDCMP is a general-purpose routine for transforming a rectangular matrix into an upper triangular matrix using orthogonal Householder transformations. QRDCMP is typically about an order of magnitude less accurate than subroutine THHCS_M, probably because it does not modify elements of $\mathbf{H}_c$ above the diagonal at each step.

#### 13.1.1.12 Subroutine SINV (ier,errm, A, n,eps,debugP)

Subroutine SINV inverts a symmetric positive-definite matrix where the upper triangular partition is stored in a vector (upper triangular by column). The subroutine first factors matrix $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ using Cholesky decomposition, then inverts $\mathbf{L}$, and finally forms $\mathbf{A} \Leftarrow \mathbf{L}^{-T}\mathbf{L}^{-1}$ as output. The Cholesky factorization is essentially the same as performed in subroutine CFACTOR, and this code could be replaced with a call to CFACTOR. As with CFACTOR, SINV also tests for a loss of precision and returns an error code indicating at which row the loss occurred. The output variable *errm* is an estimate of the retained precision in factoring $\mathbf{A}$, assuming 16-digit double precision. If the input matrix is singular, SINV returns an error. The output *ier* variable should always be checked.

#### 13.1.1.13 Subroutine SMOP (C, A,B,ip,iq,maxp,maxq)

Subroutine SMOP efficiently computes the matrix transform $\mathbf{C} = \mathbf{A}\mathbf{B}\mathbf{A}^T$, where matrix $\mathbf{A}(ip,iq)$ is a general rectangular matrix, and $\mathbf{B}(iq,iq)$ and $\mathbf{C}(ip,ip)$ are symmetric matrices stored in vectors as upper triangular by columns. No additional internal storage is required for the multiplications.

#### 13.1.1.14 Subroutine SMULT (C, A,B, n,m,l, nra,nrb,nrc)

Subroutine SMULT efficiently computes the matrix multiplication $\mathbf{C} = \mathbf{A}\mathbf{B}$, where $\mathbf{A}$ is sparse. The matrix dimensions are $\mathbf{A}(n,m), \mathbf{B}(m,l)$, and $\mathbf{C}(n,l)$. Sparse storage of $\mathbf{A}$ is not used. The sparseness of matrix $\mathbf{A}$ is tested as each row is processed, and only nonzero elements are processed. The sparseness testing only increases execution time slightly when the matrix is full and dimensions are greater than 10.

#### 13.1.1.15 Subroutine SMULT_BLOCK (C, A,B, n,m,l, nra,nrb,nrc)

Subroutine SMULT_BLOCK efficiently computes the matrix multiplication $\mathbf{C} = \mathbf{A}\mathbf{B}$ where $\mathbf{A}$ is block-wise sparse. Sparse storage of $\mathbf{A}$ is not used. The sparseness of matrix $\mathbf{A}$ is tested as each row is processed, and only nonzero elements are processed. The matrix dimensions are $\mathbf{A}(n,m), \mathbf{B}(m,l)$, and $\mathbf{C}(n,l)$.

#### 13.1.1.16 Subroutine SYMULT (c, A,b,n)

Subroutine SYMULT multiplies $\mathbf{c} = \mathbf{A}\mathbf{b}$, where $\mathbf{b}$ and $\mathbf{c}$ are vectors and $\mathbf{A}(n,n)$ is a symmetric matrix stored as upper triangular by columns. SYMULT can be called repeatedly using columns of a rectangular matrix for $\mathbf{b}$ to form a matrix multiplication.

#### 13.1.1.17 Subroutine SYSMULT (c, A,b,n)

Subroutine SYSMULT multiplies $\mathbf{c} = \mathbf{A}\mathbf{b}$, where $\mathbf{b}$ and $\mathbf{c}$ are vectors and $\mathbf{A}(n,n)$ is a symmetric matrix stored as upper triangular by columns. This differs from subroutine SYMULT in that vector $\mathbf{b}$ is assumed to be sparse. SYSMULT can be called repeatedly using columns of a rectangular matrix for $\mathbf{b}$ to form a matrix multiplication.

### 13.1.2 Kalman Filtering

#### 13.1.2.1 Subroutine CRICSS (P,E,T1,S, Ec,nd2,nd,n,lprnt)

Subroutine CRICSS solves the steady-state continuous algebraic Ricatti equation (CARE) of the form

$$\dot{\mathbf{P}} = \mathbf{G}\mathbf{Q}\mathbf{G}^T + \mathbf{F}\mathbf{P} + \mathbf{P}\mathbf{F}^T - \mathbf{P}\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{P}$$

using the Hamiltonian matrix

$$\mathbf{E}_c = \begin{bmatrix} -\mathbf{F} & -\mathbf{G}\mathbf{Q}\mathbf{G}^T \\ -\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H} & \mathbf{F}^T \end{bmatrix}.$$

The solution technique uses the matrix sign function to compute eigenvectors (see A.N. Beavers and E.D. Denman, white paper, unknown date, “Steady-State Solution to the Discrete Linear Regulator Filter and Liapunov Equation,” or Denman and Beavers 1976).

#### 13.1.2.2 Subroutine DRICSS (P,E,T1,S, Ed, nd2,nd,n,lprnt,delta)

Subroutine DRICSS solves the steady-state discrete algebraic Ricatti equation (DARE) of the form

$$\mathbf{P} = \mathbf{\Phi}\left(\mathbf{P} - \mathbf{P}\mathbf{H}^T(\mathbf{H}\mathbf{P}\mathbf{H}^T + \mathbf{R})^{-1}\mathbf{H}\mathbf{P}\right)\mathbf{\Phi}^T + \mathbf{G}\mathbf{Q}\mathbf{G}^T$$

using the Hamiltonian matrix

$$\mathbf{E}_D = \begin{bmatrix} \mathbf{\Phi}^{-1} & -\mathbf{\Phi}^{-1}\mathbf{G}\mathbf{Q}\mathbf{G}^T \\ -\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{\Phi}^{-1} & \mathbf{\Phi}^T + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{\Phi}^{-1}\mathbf{G}\mathbf{Q}\mathbf{G}^T \end{bmatrix}.$$

The solution technique uses the matrix sign function to compute eigenvectors (see A.N. Beavers and E.D. Denman, white paper, unknown date, “Steady-State Solution to the Discrete Linear Regulator Filter and Liapunov Equation,” or Denman and Beavers 1976).

#### 13.1.2.3 Module FILTERMOD

Module FILTERMOD contains variables used in various Kalman filter and smoother software. It is only necessary to set variables that are actually used in the filter code.

#### 13.1.2.4 Subroutine KF (xf,Pf,xsig, PHIt,Qt,Ht,y,sigr,t,xtr, m,mdim,n,ndim,ndyn,sparse)

Subroutine KF is a general-purpose Kalman filter that processes a single time and measurement update in one call. The state noise covariance $\mathbf{Q}_t(n,n)$ and state covariance $\mathbf{P}_f(n,n)$ matrices are stored as square for ease of use, even though only the upper triangular partition is used. (KF sets the lower triangle of $\mathbf{P}_f$ equal to the upper triangle.) KF uses variables from two modules: FILTERPREC and FILTERMOD. The precision of the filter can be set using variable *real_kind1* in FILTERPREC. Other variables in FILTERMOD control options or supply array storage. For example, variable *KFmethod* determines whether the filter measurement update is the short Kaplan form, Bierman's version of the Joseph update, or the full Joseph update.

The Kalman filter time update is

$$\begin{aligned}\hat{\mathbf{x}}_{i/i-1} &= \mathbf{\Phi}_{i,i-1}\hat{\mathbf{x}}_{i-1/i-1}\\ \mathbf{P}_{i/i-1} &= \mathbf{\Phi}_{i,i-1}\mathbf{P}_{i-1/i-1}\mathbf{\Phi}_{i,i-1}^T + \mathbf{Q}_{i,i-1}\end{aligned}$$

and the short form of the measurement update is

$$\begin{aligned}\mathbf{K}_i &= \mathbf{P}_{i/i-1}\mathbf{H}_i^T(\mathbf{H}_i\mathbf{P}_{i/i-1}\mathbf{H}_i^T + \mathbf{R}_i)^{-1}\\ \hat{\mathbf{x}}_{i/i} &= \hat{\mathbf{x}}_{i/i-1} + \mathbf{K}_i(\mathbf{y}_i - \mathbf{H}_i\hat{\mathbf{x}}_{i/i-1})\\ \mathbf{P}_{i/i} &= (\mathbf{I} - \mathbf{K}_i\mathbf{H}_i)\mathbf{P}_{i/i-1}\end{aligned} \quad .$$

### 13.1.3 Factored Filtering

The following routines are used for implementing the U-D or SRIF factored Kalman filters. Many modules are modifications of software originally written by Bierman and associates.

#### 13.1.3.1 Subroutine COV2UD_M (u,n)

Subroutine COV2UD_M is a modified version of the COV2UD subroutine in the ESP created by G.J. Bierman and R.A. Jacobson at JPL in February 1977. COV2UD_M computes the U-D factors of a positive semi-definite matrix. The input vector stored matrix is overwritten by the output U-D factors, which are also vector stored. Singular input covariances result in output matrices with zero columns. The modifications in COV2UD_M are listed in the prolog. Note that module FILTERPREC defines the floating point precision.

#### 13.1.3.2 Subroutine PHIU_M (Pu, PHI,nph,nr,nc,U,nu,npu)

Subroutine PHIU_M computes $\mathbf{P}_u = \mathbf{\Phi}\mathbf{U}$ where $\mathbf{\Phi}(nr,nc)$ is a sparse, 2-dimensional matrix and $\mathbf{U}(nc(nc+1)/2)$ is unit upper triangular. Although this routine has a similar function as the ESP subroutine PHIU written by Bierman and Nead at JPL in February 1978, it was developed independently and is designed to search for and only process nonzero elements in $\mathbf{\Phi}$.

#### 13.1.3.3 Subroutine R2A_M (A, R,n,namr,imaxa,ja,nama)

Subroutine R2A_M copies the triangular vector stored matrix $\mathbf{R}$ into matrix $\mathbf{A}$ and arranges the columns to match the desired *nama* parameter list. Names in the *nama* list that do not correspond to any name in the *namr* list have zero entries in the corresponding $\mathbf{A}$ column.

R2A_M is a modification of the ESP subroutine R2A written by Bierman and Nead at JPL in September 1976, where modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision.

#### 13.1.3.4 Subroutine RANK1_M (Uout,ierr, v, Uin,n,c)

Subroutine RANK1_M is a stable U-D factor rank 1 update of the form $\mathbf{U}_{out}\mathbf{D}_{out}\mathbf{U}_{out}^T = \mathbf{U}_{in}\mathbf{D}_{in}\mathbf{U}_{in}^T + c\,\mathbf{v}\mathbf{v}^T$, where $\mathbf{U}_{in}, \mathbf{U}_{out}$ are unit upper triangular matrices, $\mathbf{D}_{in}, \mathbf{D}_{out}$ are diagonal matrices, $c$ is a scalar, and $\mathbf{v}$ is a vector. This "_M" version is a modification of the original Bierman/Nead ESP routine RANK1, where modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision.

#### 13.1.3.5 Subroutine RDX (del, R,dx,n)

Subroutine RDX computes $\mathbf{d}_{el} = \mathbf{R}\,\mathbf{d}_x$ where $\mathbf{R}(n(n+1)/2)$ is the upper triangular square-root information matrix and $\mathbf{d}_x$ is a column vector.

#### 13.1.3.6 Subroutine RINCON_M (Rout, cnb, Rin,n)

Subroutine RINCON_M computes $\mathbf{R}_{out} = \mathbf{R}_{in}^{-1}$, where $\mathbf{R}_{in}$ is an upper triangular vector stored input matrix and $\mathbf{R}_{out}$ is also upper triangular. Using $\mathbf{R}_{in} = \mathbf{R}_{out}$ in the call is permitted. If the input *cnb* is nonnegative, RINCON_M also computes a condition number estimate $cnb = \|\mathbf{R}\|_F \, \|\mathbf{R}^{-1}\|_F$ where

$$\|\mathbf{R}\|_F = \left( \sum_{i=1}^{m} \sum_{j=1}^{n} C_{ij}^2 \right)^{1/2}$$

is the Frobenius norm. Output *cnb* is a bound on the true condition number: $cnb / n \le$ condition number $\le cnb \cdot n$ (see Lawson and Hanson 1974).

When $\mathbf{R}_{in}$ is singular, RINCON_M computes an inverse corresponding to the system having zero rows and columns associated with the singular diagonal entries. This "_M" version is a modification of the Bierman/Nead ESP routine RINCON, where modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision.

#### 13.1.3.7 Subroutine RINZ_M (x,ierr, R,n,z)

Subroutine RINZ_M computes the solution vector $\mathbf{x}$ to the matrix equation $\mathbf{R}\mathbf{x} = \mathbf{z}$ by back substitution, where $\mathbf{R}(n(n+1)/2)$ is a vector-stored upper triangular matrix. Computation is minimized and accuracy enhanced by not computing the matrix inverse. If $\mathbf{R}$ is singular, with its *j*-th diagonal element as its last diagonal zero value, $x(i)$ is computed only for $j < i \le n$. The other components of $x$ are left unchanged for $1 \le i \le j$, and the value of $j$ is returned as *ierr*. Subroutine RINZ_M is a modification of the Bierman ESL routine RINZ, where modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision.

#### 13.1.3.8 Subroutine RI2COV_M (sig,Covout, Rinv,n,krow,kcol)

Subroutine RI2COV_M computes the standard deviations and, if desired, the covariance matrix ($\mathbf{C}_{ovout} = \mathbf{R}^{-1}\mathbf{R}^{-T}$) of a vector-stored upper triangular square-root information matrix. The output covariance matrix is also vector-stored. This "_M" version is a modification of the original Bierman/Nead ESP routine RI2COV, where modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision.

#### 13.1.3.9 Subroutine RTINZ (ier,x, R,n,y)

Subroutine RTINZ solves the equation $\mathbf{R}^T\mathbf{x} = \mathbf{y}$ for vector $\mathbf{x}$ by forward substitution where $\mathbf{R}$ is upper triangular. If any diagonal of $\mathbf{R}$ is zero, error return flag *ier* is set.

#### 13.1.3.10 Subroutine SRIF (xfs,Rsrif,xsig, PHIt,qt,Ht,y,sigr,t,m,mdim,n,ndim,ndyn,sparse)

Subroutine SRIF computes a single time and measurement update of the square root information filter. The subroutine uses variables from two modules: FILTERPREC and FILTERMOD. The precision of the filter can be set using variable *real_kind1* in FILTERPREC. Other variables in FILTERMOD control options or provide array storage.

#### 13.1.3.11 Subroutine TDHHT_M (S,v, imaxs,irs,jcs,jstart,jstop)

Subroutine TDHHT_M transforms a rectangular double-subscripted matrix $\mathbf{S}(irs, jcs)$ to an upper triangular or partially upper triangular form using Householder orthogonal transformations. It is assumed that the first *jstart-1* columns of $\mathbf{S}$ are already triangularized. This "_M" version is a modification of the Bierman ESL routine TDHHT, where modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision. TDHHT_M also tests for sparseness in matrix **S** and only operates on nonzero elements.

#### 13.1.3.12 Subroutine THHC_M (R,Hc,rsos, n,mmax,m,nstrt,sparse)

Subroutine THHC_M is a general purpose routine for transforming an upper triangular plus a rectangular 2-D matrix into an upper triangular matrix using orthogonal Householder transformations. THHC_M is based on the THH subroutine in the ESP written by Bierman and Hamata at JPL in March 1978. Modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision. THHC_M also tests for sparseness in matrix **Hc** and only operates on nonzero elements

#### 13.1.3.13 Subroutine TRIPHU (U,v, a,vd, nda,n,d)

Subroutine TRIPHU computes $\mathbf{U}$ such that $\mathbf{U}\mathbf{D}_1\mathbf{U}^T = \mathbf{A}\mathbf{D}\mathbf{A}^T$, where $\mathbf{U}$ is upper triangular. It also updates $\mathbf{A}$ such that $\mathbf{A}\mathbf{A}^T \Leftarrow \mathbf{A}\mathbf{D}\mathbf{A}^T$. Subroutine TRIPHU first forms $\mathbf{A}' = \mathbf{A}\mathbf{D}^{1/2}$ and then triangularizes $\mathbf{A}'$ using post-multiplying Householder transformations such that $(\mathbf{A}'\mathbf{T})(\mathbf{T}^T\mathbf{A}'^T) = \mathbf{A}\mathbf{D}\mathbf{A}^T$, where $\mathbf{T}\mathbf{T}^T = \mathbf{I}$ and $\mathbf{A}'\mathbf{T}$ is upper triangular. The final step is conversion to $\mathbf{U}\mathbf{D}_1\mathbf{U}^T$ where $\mathbf{U}$ is unit upper triangular. Typically on input $\mathbf{A} = \mathbf{\Phi}\mathbf{U}$. TRIPHU uses module FILTERPREC to define floating point precision.

#### 13.1.3.14 Subroutine UDKF (xfu,UD,xsig, PHIt,Qt,Ht,y,sigr,t,m,mdim,n,ndim,ndyn,sparse)

Subroutine UDKF computes a single time and measurements update of the U-D filter. It can take advantage of sparseness in $\mathbf{\Phi}$ when computing the time update. UDKF uses variables from two modules: FILTERPREC and FILTERMOD. The precision of the filter can be set using variable *real_kind1* in FILTERPREC. Other variables in FILTERMOD control options or provide array storage.

#### 13.1.3.15 Module UDMES_M

Module UDMES_M computes the state estimate and U-D measurement-updated covariance, $\mathbf{P} = \mathbf{UDU}^T$. This version is a modification of the original Bierman/Nead ESP routine UDMEAS, where modifications are listed in the prolog. The original UDMEAS was split into UDMES1 and UDMES2 so that the measurement outlier test could be performed after executing UDMES1. Then UDMES2 is bypassed if the measurement is an outlier. UDMES1 output is useful for some applications. Note that module FILTERPREC defines the floating point precision.

#### 13.1.3.16 Subroutine UD2COV_M (Pout, Uin,n)

Subroutine UD2COV_M computes a covariance from its U-D factorization ($\mathbf{P}_{out} = \mathbf{UDU}^T$). Both $\mathbf{P}_{out}$ and $\mathbf{U}_{in}$ are vector-stored and the output covariance can overwrite the input $\mathbf{U}_{in}$ array. This "_M" version is a modification of the original Bierman/ Nead ESP routine, where modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision.

#### 13.1.3.17 Subroutine UD2SIG_M (sig, u,n,sf,jstrt,nsig)

Subroutine UD2SIG_M computes the standard deviations (sigmas) from U-D covariance factors for rows $jstrt$ to $\min(jstrt + nsig - 1, n)$. UD2SIG_M is based on the Bierman/Nead ESP subroutine UD2SIG, where modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision.

#### 13.1.3.18 Subroutine WGS_M (U,v, W, imaxw,iw,jw,dw,sparse)

Subroutine WGS_M is a modified weighted Gram-Schmidt algorithm for reducing $\mathbf{W}\mathbf{D}_w\mathbf{W}^T$ to $\mathbf{UDU}^T$ where $\mathbf{U}$ is a vector-stored upper triangular matrix with $\mathbf{D}$ elements stored on the diagonal. This "_M" version is a modification of the original Bierman/Nead ESP routine, where modifications are listed in the prolog. Note that the calling argument order has been changed, and module FILTERPREC defines the floating point precision. Also WGS_M tests for sparseness in matrix $\mathbf{W}$ and only operates on nonzero elements.

### 13.1.4 General Functions

The following three modules have multiple applications.

#### 13.1.4.1 Subroutine ERRHAND (message)

Subroutine ERRHAND writes an error message to Fortran unit 6 and stops execution with a "stop error" message.

#### 13.1.4.2 Function RNOR (jd)

Real(8) function RNOR generates quasi-normal random numbers with zero mean and unit standard deviation. It can be used with any computer with integers at least as large as 32767. The first RNOR call should set *jd* to any nonzero integer. This initializes RNOR and the first random number is returned. On subsequent calls $z = rnor(0)$ causes the next random number to be returned. RNOR was written by D. Kahaner at Scientific Computing Division, NBS, and G. Marsaglia at Computer Science Department, Washington State University in 1981 (Marsaglia and Tsang 1983).

#### 13.1.4.3 Function UNI (jd)

Real(8) function UNI generates quasi-uniform random numbers on [0,1). It can be used on any computer that allows integers at least as large as 32,767. The first UNI call should set *jd* to any nonzero integer. This initializes the program and the first random number is returned. On subsequent calls $z = uni(0)$ causes the next random number to be returned. UNI was written by J. Blue and D. Kahaner at Scientific Computing Division, NBS, and G. Marsaglia at Computer Science Department, Washington State University in 1981 (G. Marsaglia, "Comments on the perfect uniform random number generator", unpublished notes, Washington State University 1981).

### 13.1.5 Spectral Analysis and Autoregressive Moving Average (ARMA) Modeling

#### 13.1.5.1 Subroutine BT_SPECT (pwr,delf,n3, c,nc,dt,nlag,iWindow)

Subroutine BT_SPECT computes the power spectrum of real(4) data vector **c** using the Blackman-Tukey correlogram implemented using the *fast Fourier transform* (FFT) with tapered windows. Variable *iWindow* specifies use of either Bartlett (1) or Hamming (2) window functions.

#### 13.1.5.2 Subroutine CVA (PHI,H,Q,Rn,Bw,Gu,Ac,n, nyi,nyui,mi,minlag,maxlag,ndim,yu,eps,dt,directUfeed)

Subroutine CVA performs canonical variate analysis on real(8) data array $\mathbf{yu}(n_{yui}, m_i)$, which contains both measurements $\mathbf{y}(n_{yi}, m_i)$ and optional exogenous (control) inputs $\mathbf{u}(n_{ui}, m_i)$. The model output consists of arrays for the discrete state-space model

$$\begin{aligned} \mathbf{y}_i &= \mathbf{H}\mathbf{x}_i + \mathbf{A}\,\mathbf{u}_i + \mathbf{B}\mathbf{q}_i + \mathbf{r}_i \\ \mathbf{x}_{i+1} &= \mathbf{\Phi}\mathbf{x}_i + \mathbf{G}\mathbf{u}_i + \mathbf{q}_i \end{aligned}.$$

A "contained" subroutine PLOTPSD2 allows computation of the power spectral density (PSD). Subroutine CVA uses the methods outlined by W. Larimore (1983, 1987).

#### 13.1.5.3 Subroutine DFT (c,d, isn,n,a,b)

Subroutine DFT computes a discrete Fourier transform of (real, imaginary) data ($\mathbf{a},\mathbf{b}$) in single precision (real(4)). The output (real, imaginary) transform is in ($\mathbf{c},\mathbf{d}$). Variable *isn* controls the transform direction. DFT uses a brute-force approach, not the FFT. However trigonometric functions are computed recursively to reduce computation.

#### 13.1.5.4 Subroutine FFT_MIXEDRADIX (a,b, ntot,n,nspan,isn)

Subroutine FFT_MIXEDRADIX computes the mixed-radix complex FFT using an algorithm developed by R. C. Singleton, Stanford Research Institute, October 1968 (Singleton 1969). The routine can also be used for multivariate data. Contained subroutine REALTR allows more efficient computation when the data are real and the number of points is divisible by two. Subroutine FFT_MIXEDRADIX is a greatly modified version of the original Singleton subroutine. The modifications are: (1) create subroutines from code segments, (2) minimize use of "go to" statements, (3) use Fortran 90 features (intent, do while, etc.), and (4) include common variables in module FFTCOM. The modified code was tested by comparison with subroutine DFT and the original FFT subroutine using all even numbers of data points from 100 to 500. (Approximately 50% were rejected as not meeting radix factor requirements.)

#### 13.1.5.5 Subroutine MARPLE_SPECT (pwrDB,delf, nc,mmax,nplot,x,dt,tol1,tol2)

Subroutine MARPLE_SPECT computes the power spectrum of real(4) data vector **x** using Marple's autoregressive (AR) spectral analysis approach. The method efficiently computes AR coefficients that minimize the squared prediction error of forward and backward prediction filters without using the Levinson recursion constraint. The model order is determined using prediction error tolerances rather than the *final prediction error* (FPE) of Burg's *maximum entropy method* (MEM). The PSD is calculated from the computed AR coefficients and noise power. Compared to the Burg and Yule-Walker (YW) MEM approaches, the algorithm reduces bias in frequency estimates of spectral components, reduces variance in frequency estimates over an ensemble of spectra, and does not "split" spectral lines when working with very narrow band signals (Marple 1980).

#### 13.1.5.6 Subroutine MEMOPT (delf,pwrDB, f,m,npmin,npmax,nlag,dt,iunit,junit,nplt)

Subroutine MEMOPT computes the power spectrum of real(4) data vector **x** using the Burg MEM. The Burg MEM computes AR coefficients that minimize the squared prediction error of forward and backward prediction filters. MEMOPT first fits models and computes the Akaike final prediction error (FPE) for all model orders up to the maximum *npmax*. It then recomputes the model for the order with minimum FPE and uses the AR coefficients to compute the PSD (Ulrych and Bishop 1975).

#### 13.1.5.7 Subroutine SPECT (d,delf,n3, c,nc,nplt,junit,dt,nseg,iWindow)

Subroutine SPECT computes the periodogram power spectrum of real(4) data vector **c** using the FFT with optional tapered windows. The FFT is Singleton's mixed-radix algorithm that works with many even numbers of data points $n_c$. If the initial number $n_c$ is not acceptable for use in the FFT, SPECT will reduce $n_c$ by increments of two until an acceptable number is found. Variable *iWindow* specifies use of either rectangular (0), 10% cosine taper (1), Hamming (2), or Welch 50% overlapping segments using the Bartlett window (3).

#### 13.1.5.8 Subroutine SPECTD (d,delf,n3, c,nc,nplt,junit,dt,nseg,iWindow)

Subroutine SPECTD computes the periodogram power spectrum of real(4) data vector **c** using a DFT with optional tapered windows. Subroutine SPECTD is essentially the same as subroutine SPECT except that it uses the DFT (with any number of data points) rather than FFT. Variable *iWindow* specifies use of either rectangular (0), 10% cosine taper (1), Hamming (2), or Welch 50% overlapping segments using the Bartlett window (3).

#### 13.1.5.9 Subroutine YW_SPECT (pwrDB,delf,g, x,nc,dt,nlag,nplot)

Subroutine YW_SPECT computes the power spectrum of real(4) data vector **x** using the YW MEM. The YW normal equations (composed of sample autocorrelation coefficients) are solved to estimate coefficients of an AR prediction error filter. YW_SPECT first fits models and computes the Akaike FPE for all model orders up to the maximum *nlag*. It then recomputes the model for the order with minimum FPE and uses the AR coefficients to compute the PSD (Ulrych and Bishop 1975).

### 13.1.6 Numerical Optimization

#### 13.1.6.1 Module OPTLDAT

Module OPTLDAT includes model variables for the FUNCT subroutine called by the optimization routines. Other variables may be added to OPTLDAT as needed for specific models.

#### 13.1.6.2 Subroutine OPTL_BACK (cost,R,g,nmeas,irflag, xtot, nxtot,niter,xmin,xmax,convtst,lprnt_o)

Subroutine OPTL_BACK computes the state estimate **xtot**(*nxtot*) minimizing a residual sum-of-squares cost function. The optimal state estimate is obtained using Gauss-Newton iterations with backtracking to handle nonlinearities. It also handles minimum/maximum state constraints. A user-supplied function

FUNCT (cost,pf,r,g,delx,nmeas, partial,xtot,nxtot,debugp)

must be provided, where variables (*partial,xtot,nxtot,debugp*) are inputs and (*cost,pf,r,g,delx,nmeas*) are outputs. These variables are defined in comments in OPTL_BACK. Also see the *Chapter7/passiveTrk/passTrk.f* code for an example of usage.

#### 13.1.6.3 Subroutine OPTL_LM (cost,nmeas,istat, xest,Pf, nest,niter,convtst,lambda0,maxLambda,lprnt_o)

Subroutine OPTL_LM computes the state estimate **xest**(*nest*) minimizing a residual sum-of-squares cost function. The optimal state estimate is obtained using Gauss-Newton iterations with Levenburg-Marquard optimization logic to handle nonlinearities. A user-supplied function

FUNCT (cost,pf,ainf,b,dxn,nmeas, partial,xest,nest,debugp)

must be provided where variables (*partial,xest,nest,debugp*) are inputs and (*cost,pf,ainf,b,dxn,nmeas*) are outputs. These variables are defined in comments in OPTL_LM. Also see the *Chapter7/passiveTrk/passTrk.f* code for an example of usage.

## 13.2 Chapter 3: Modeling Examples

### 13.2.1 Program INS_SIM

Program INS_SIM generates simulated strapdown inertial navigation system position error measurements as described in Section 3.3. It also simulates a missile trajectory. The output data files are "taimes.dat" (position measurement data), and "specficForce.plt" (specific force vs. time). These files are used in program TST_INS to test Friedland's bias-free/bias-restoring filter, and in program TST_IMU_FILTERS to compare numerical accuracy of filter implementations (Section 10.4). The files are also used in Example 11.3 to demonstrate jump detection.

## 13.3 Chapter 5: Linear Least-squares Estimation Solution Techniques

This directory contains several programs used to compare linear least-squares solutions.

### 13.3.1 Program TST_INVERT

Program TST_INVERT tests the accuracy of several least-squares solution techniques for a polynomial model, which is known to stress numerical accuracy. The methods tested include:

1. Normal equations and Cholesky factorization/inversion (subroutines SINV and SYMULT)
2. Normal equations and Gaussian-Jordan elimination/inversion (subroutine MATINV_GJ)
3. MGS QR decomposition and inversion (subroutines THHC_M, RINZ, RINCON_M)
4. Singular value decomposition (LAPACK subroutines DGESDD or DGESVD)
5. Conjugate gradient on the normal equations to reduce the residual (subroutine CGNR)
6. LSQR (subroutine LSQR)

The results from this program are used in Examples 5.4, 5.8, 5.9, 5.10, 5.11, and 5.13. Output plot files are:

1. tstMont_*.plt: includes Monte Carlo sample error statistics for the different methods
2. tstCov_*.plt: includes covariance error statistics for the different methods
3. predictErr_*.plt: includes the polynomial model prediction errors for the different methods

### 13.3.2 Program TIME_INVERT

Program TIME_INVERT compares the timing of different least-squares algorithms for a polynomial model. The methods tested include:

1. Normal equations and Cholesky factorization/inversion (subroutines SINV and SYMULT)
2. Normal equations and Gaussian-Jordan elimination/inversion (subroutine MATINV_GJ)
3. MGS QR decomposition and inversion (subroutines THHC_M, RINZ, RINCON_M)
4. Singular value decomposition (LAPACK subroutines DGESDD or DGESVD)

The results from this program are used in Section 5.6.2.

### 13.3.3 Program Time_INVERT2

Program Time_INVERT2 compares the timing of different least-squares algorithms for a 500 point time-correlated model. The methods tested include:

1. Normal equations and Cholesky factorization/inversion (subroutines SINV and SYMULT)
2. Normal equations and Gaussian-Jordan elimination/inversion (subroutine MATINV_GJ)
3. MGS QR decomposition and inversion (subroutines THHC_M, RINZ, RINCON_M)
4. Singular value decomposition (LAPACK subroutines DGESDD or DGESVD)

The results from this program are used in Section 5.6.2.

### 13.3.4 Program TST_CHEBV

Program TST_CHEBV compares the timing of different least-squares algorithms for a Chebyshev polynomial model. The only methods tested are the normal equations and Cholesky factorization/inversion. The results from this program are used in Figure 5.11.

## *13.4 Chapter 6: Model Errors and Model Order*

### 13.4.1 Program IMAG_MISALIGN

Program IMAG_MISALIGN models misalignments in an optical imaging instrument, and it tests the accuracy of reduced-order models for least-squares estimation. Singular value decomposition of the information matrix is used to determine observable linear combinations. Fit accuracy is evaluated using Monte Carlo simulation. Stepwise regression is used to compute a reduced-order model. This is used in Examples 6.6 and 6.7. Subroutines in files IMAG_LIB.F and JUMPCOM.FOR are also required.

### 13.4.2 Program LS_COV

Program LS_COV is a general-purpose program that computes the total Bayesian least-squares error covariance (including effects of consider parameters) using equations listed in Section 6.2. The selection of adjusted, consider, and ignored states is controlled by a text file listing the status and prior information for each parameter. LS_COV calls module FILTERMOD to obtain the dynamic and measurement model information, and LSMODEL to define the state and measurement model. Driver program RUN_LSCOV executes LS_COV for a 5th order polynomial model.

### 13.4.3 Program MONTEPOLY_SUBOPT

Program MONTEPOLY_SUBOPT computes (using Monte Carlo simulation) the sample covariance and fit/prediction residuals of suboptimal least-squares estimates for a polynomial model. It generates output plot files sigyMont.plt and sigx.plt. This is used in Example 6.5.

### 13.4.4 Program PLOTCOV

Program PLOTCOV generates a 3-D wire mesh plot file for a specified covariance matrix, as demonstrated in Example 6.4. This version takes input in earth-centered inertial (ECI) coordinates and generates output in both ECI and geodetic latitude, longitude and altitude coordinates.

### 13.4.5 Program TST_POLY

Program TST_POLY generates simulated measurements for a polynomial model and computes optimal or reduced-order least-squares fits to the data. Both the normal equations and SVD solutions are used. The residuals and the residuals covariance are written to plot files resid04.plt, covy04.plt, and covyIndx04.plt. This is used in Example 6.1.

### 13.4.6 Program TST_POLY_SUBOPT

Program TST_POLY_SUBOPT computes the fit/prediction residual covariance of a suboptimal least-squares estimate for a polynomial model. It generates output plot files sigy.plt. This is used in Example 6.5.

## *13.5 Chapter 7: Least Square Estimation – Constraints, Nonlinear Models and Robust Techniques*

### 13.5.1 Program MONTEPOLY_CONSTRT

Program MONTEPOLY_CONSTRT tests the numerical accuracy of different methods for constrained least-squares estimation. The Monte Carlo simulation uses an 11th-order polynomial model. The RMS errors are listed in Example 7.1 for the following constrained least-squares methods:

1. normal equations with constraints as measurements,
2. unconstrained MGS QR using Lagrange multipliers,
3. MGS QR with constraints as measurements,
4. LAPACK GSVD (generalized SVD),
5. LAPACK GSVD with explicit coding of constraints.

### 13.5.2 Program QUAD_COST

Program QUAD_COST uses a mildly quadratic function to demonstrate convergence of Newton and Gauss-Newton iterations for finding the minimum of the cost function. The program creates output plot file "quad.plt " for use in Example 7.2.

### 13.5.3 Program NONLINEAR_BALL

Program NONLINEAR_BALL simulates range tracking of a falling ball with unknown initial location, and compares convergence of Gauss-Newton and Newton iterations for nonlinear least-squares solutions. It generates plot files "problem.plt" (problem geometry) and "cost.plt" (cost versus x and y) for use in Example 7.3. The output report file is called "ball.lis"

### 13.5.4 Program PASSTRK

Program PASSTRK tests the convergence of four nonlinear least-squares solution techniques for passive ship tracking, and it computes a table grid of the true least-squares cost function, the Fisher information (quadratic model) cost function, and the Hessian quadratic model of the cost function versus the four epoch states of position and velocity. The four user-selectable optimization algorithms are Levenberg-Marquard, backtracking, backtracking followed by secant iterations, and the NL2SOL algorithm. PASSTRK generates three output plot files for use in Example 7.4: "passTrk.plt" is the "own ship" and "target ship" position versus time, "costp.plt" contains the true, Fisher and Hessian cost versus position given the least-squares estimate of velocity, and "covtv.plt" contains the true, Fisher and Hessian cost versus velocity given the least-squares estimate of position. The plot files also contain, as a separate zone, the position and velocity states for each step of the selected algorithm.

## 13.6 Chapter 8: Kalman Filtering

### 13.6.1 Program TST_KF1

Program TST_KF1 simulates measurements for a first-order Markov process model, and computes the *a posteriori* variance and filter estimates for a one-state Kalman filter. Three different levels (0.1, 1.0, 10) of process noise variance Q are tested, with measurement noise variance R =100. The output plot file is "kf1.plt". This is used for example 8.1.

### 13.6.2 Program TST_KF2

Program TST_KF2 simulates measurements for a two-state system, and computes the *a posteriori* variance and filter estimates for a two-state Kalman filter. Process noise drives only the second state of the continuous system, and three different levels (0.1, 1.0, 10) of process noise PSD are tested. State #1 is the integral of #2, and only state #1 is measured. The measurement noise PSD is 100. The output plot file is "kf2.plt". This is used for example 8.2.

### 13.6.3 Program TST_KF2CORR

Program TST_KF2CORR simulates time-correlated (correlation = 0.5) measurements for a two-state continuous system, and computes the *a posteriori* variance and state estimates using different methods for handling time-correlated measurements in the Kalman filter. The four methods are the standard two-state Kalman filter, a three-state Kalman filter with the added state modeling the correlation, a three-state Kalman filter where the added state is a delay state, and a two-state Bryson-Hendrikson approach. Process noise with PSD of 1.0 drives only the second state of the continuous system. State #1 is the integral of #2, and only state #1 is measured. The measurement noise PSD is 100. The output plot file is "kf2corr.plt". This is used for example 8.3.

### 13.6.4 Program TST_KF2CONT

Program TST_KF2CONT simulates measurements for a two-state continuous system, and computes the *a posteriori* variance and filter estimates for a two-state continuous Kalman filter. Process noise drives only the second state of the continuous system, and three different levels (0.1, 1.0, 10) of process noise PSD are tested. State #1 is the integral of #2, and only state #1 is measured. The measurement noise PSD is 100. The output plot file is "kf2cont.plt". This is used for Example 8.3.

### 13.6.5 Program STEADY_STATEKF6

Program STEADY_STATEKF6 computes the solution to steady-state continuous and discrete Kalman filters for a six-state position-velocity-acceleration problem. The different solution methods are (1) execute a standard KF for a "long" time, (2) use Schur decomposition of a Hamiltonian matrix, or (3) use the matrix sign function to compute eigenvalues of a Hamiltonian matrix. The output file is "ssKF6.lis". The results are used in Example 8.5.

### 13.6.6 Program WIENER

Program WIENER computes the Wiener filter and steady-state Kalman filter solution for a one-state Markov process problem. The output file is "wkf.lis". The results are used to validate equations in Example 8.6.

### 13.6.7 Program TST_INS

Program TST_INS compares results from the Friedland bias-free/bias-restoring filter described in Section 8.4.1 with the standard Kalman filter. This uses measurement file "TAImes.dat" generated by program INS_SIM.

## *13.7 Chapter 9: Filtering for Nonlinear Systems, Smoothing, Error Analysis/Model Design, and Measurement Preprocessing*

### 13.7.1 Program CONDMEAN2

Program CONDMEAN2 generates a plot file of the conditional probability distribution for the one-state nonlinear model $y = x + \alpha x^2 + r$ for five values of measurement noise ($r = -1.5, -0.75, 0, 0.75, 1.5$). It also tests extended Kalman filter (EKF) and iterated extended Kalman filter (IEKF) solutions for 10000 time points with simulated N(0,1) values for $r$ Plot files "condProb.plt" lists the conditional probability distribution versus $x$ for each $r$-value, and "EKF.plt" lists the state estimate and $1-\sigma$ uncertainty versus time as computed by the EKF and IEKF. This is used in Examples 9.1 and 9.2.

### 13.7.2 Module COVAR_ANAL

Module COVAR_ANAL is a general-purpose module that performs Kalman filter error analysis using equations listed in Chapter 9. It is similar to INS_ERROR but is not restricted to a single model. The selection of adjusted, consider, and ignored states is controlled by a text file listing the status and prior information for each parameter. COVAR_ANAL calls module FILTERMOD to obtain the dynamic and measurement model information.

### 13.7.3 Program TST_KFSMO

Program TST_KFSMO tests three filter-smoothers (forward-backward, RTS, and mBF) using the two-state problem $x_1(t_i) = x_1(t_{i-1}) + (t_i - t_{i-1})\,x_2(t_{i-1}),\ x_2(t_i) = x_2(t_{i-1}) + q(t_i)$ where $q(t_i)$ is $N(0,1)$ noise. The measurement is $y(t_i) = x_1(t_i) + r(t_i)$ where $r(t_i)$ is $N(0{,}500)$ when $t_i - t_{i-1} = 0.2$. Plot file "smooth.plt" lists the state estimate and $1-\sigma$ uncertainty versus time as computed by the three smoothers. Plot file "xerr_smo.plt" lists the error in the RTS smoother state estimates versus time, and "Smo_sig.plt" lists the RTS smoother $1-\sigma$ uncertainty. A scratch binary file is also used on Fortran unit 42. The results for the forward-backward smoother are used in Examples 9.3.

### 13.7.4 Subroutine FILTER_SMOOTH

Subroutine FILTER_SMOOTH includes subroutines to implement a general purpose Kalman filter and RTS smoother. It is implemented using sparse matrix multiplication functions for reduced execution time on large problems. FILTER_SMOOTH is the driver for the filter-smoother calculations, and it calls module MODELMOD to obtain the dynamic and measurement model information. The supplied version of MODELMOD.F90 duplicates the 8-state model used in ACTRK_FILSMO. The measurement data file (ACMEAS.TXT) was generated by ACTRK_FILSMO.

The selection of parameters to be estimated is controlled by a file listing the status and prior information for filter states and parameters. The "keyword" input text file sets control parameters, and initial state values. The acceptable keyword options are listed in the comments within FILTER_SMOOTH.

### 13.7.5 Program TANK_FILSMO

Program TANK_FILSMO is a maneuvering tank-tracking six-state EKF-filter/RTS-smoother using the reconstructed ATMT Scout tank data as described in Section 3.2.1. The input data file is “scoutSim4.plt”. Output plot files “Xerr_ekfsTnk.plt” and “Xerr_rtsTnk.plt” list the estimate errors for the filter and smoother respectively. The output report file contains state estimates at various locations in the code. This is used in Example 9.4.

### 13.7.6 Program ACTRK_FILSMO

Program ACTRK_FILSMO is an eight-state EKF-filter/RTS-smoother that tracks a simulated aircraft using the model described in Section 3.2.2. Aircraft maneuvers are defined in input file “mnvs.dat”. Output plot files “ACTrk.plt” lists the true (simulated) states versus time, “Filt_Xerr.plt” lists the filter state estimate errors and “Filt_sig.plt” lists the $1-\sigma$ uncertainty. Files “xerr_smo.plt” and “Smo_sig.plt” are the comparable outputs generated by the smoother. This is used in Example 9.5.

### 13.7.7 Program INS_ERROR

Program INS_ERROR analyzes the effects of five error sources on Kalman filter estimates using the approach described in Section 9.3.1. The five error sources are initial condition errors, un-estimated bias parameters, consider bias parameters, measurement noise, and process noise. The simulated system is the 27-state model for errors of an inertial navigation system as described in Section 3.3. The input measurement file “taimes.dat” is generated for a simulated missile flight using program SIM_INS. Text file “taiState.ctl” defines the state names, initial values, initial $1-\sigma$ uncertainty, process noise, and status (unadjusted, consider, adjusted). The output plot files are “filtErr.plt” (filter estimate errors versus time as generated by a standard Kalman filter), “KFcovarEst.plt” (“error analysis filter” state estimates), “KFcovarErr.plt” (error in “error analysis filter” state estimates), and “filterSig.plt” (breakdown of $1-\sigma$ filter uncertainty for each of the five error sources.). Program INS_ERROR is used in Example 9.7. It calls module COVAR_ANAL, which contains various subroutines used for the error analysis. COVAR_ANAL is structured to be general-purpose and thus may be used for other systems.

### 13.7.8 Program PASSTRK_FIL

Program PASSTRK_FIL tests the linearized Kalman filter, EKF and IEKF solutions for passive ship tracking. The output plot files are “passTrk.plt” (own and target ship positions versus time), “Xerr_ekf.plt“ (estimate errors for EKF), “Xerr_lkf.plt” (estimate errors for linearized KF), and

"Xerr_iekf.plt" (estimate errors for IEKF). Results are discussed in Section 9.1.2, but results are not shown because all methods performed poorly.

## 13.8 Chapter 10: Factored (Square Root) Filtering

### 13.8.1 Program TST_SINGLEPREC

Program TST_SINGLEPREC tests the accuracy of one-state, one-measurement Kalman filters (standard and Joseph forms) implemented in single precision. It must be compiled without optimization to properly test the loss of accuracy.

### 13.8.2 Program TST_IMU_FILTERS

Program TST_IMU_FILTERS tests the numeric accuracy and timing of the covariance Kalman filter, U-D filter and SRIF on the 27-state INS example. The *runFilt*(3) logical variables determines whether the Kalman, U-D and SRIF are to be executed. (All may operate in parallel if desired, but this is not done when timing the algorithms.) The *KFmethod* variable determines whether the short Kalman, Bierman's version of the Joseph measurement update, or the full Joseph covariance update are to be used for the covariance Kalman filter. The input measurement file "taimes.dat" is generated for a simulated missile flight using program SIM_INS. Text file "taiState.ctl" defines the state names, initial values, initial $1-\sigma$ uncertainty, process noise, and status (unadjusted, consider, adjusted). Input file "xestUD8_1.dat" (or similar name) contains the state estimates from a U-D filter that was executed in double precision. It is used as a reference for the other filters. The output plot files are "KFEst.plt" (standard covariance Kalman filter estimates vs. time), "KFErr.plt" (error in filter estimates vs. time for all filters executed), "KFsig.plt" ($1-\sigma$ uncertainty vs. time for all filters executed). Program TST_IMU_FILTERS is used for examples in Section 10.4. In addition to library subroutines, FILTERS must be linked because it is the driver for the filters.

### 13.8.3 Subroutine FILTERS

Subroutine FILTERS is a high-level driver for the Kalman filters used in program TST_IMU_FILTERS. It is general-purpose and may be used with other models.

## 13.9 Chapter 11: Advanced Filtering Topics

### 13.9.1 Program TST_FIS2

Program TST_FIS2 tests the calculation of the Fisher information matrix for a four-state problem. It can also run scoring iterations. Module MODELMOD contains the model-specific information and TST_FIS2 is the driver for the filter calculations. The four-state model consists of two independent first-order Markov processes with one state as the integral of the Markov process. Measurements may be either independent states $x$ and $y$ (*mmod*=2), or range and bearing (*mmod*=1). The six parameters to be estimated via MLE are Markov process constants (2), Markov process noise (2), and measurement noise (2).

Search for "###" to find control parameters that are typically changed. If *test_fis* =.true., the gradient and Fisher information matrix are computed using numeric central-difference partial derivatives, and the average Fisher is computed using 1000-sample Monte Carlo simulation. If *nsamp* = 1, the numeric gradient and Fisher can be compared with the analytic gradient/Fisher (when *test_fis* =.false.) to verify coding.

The filter/simulation is coded with the following assumptions:

1. the initial true state is (1,10,0,0),
2. the filter state is initialized at truth, with diagonal covariance,
3. $\mathbf{\Phi}$ is analytic, but $\mathbf{Q}$ is assumed diagonal because the sampling rate is high compared with Markov process time constants. Random process noise is applied independently to the two channels,
4. measurement noises of the two measurements are independent,
5. the measurement model does not include unknown parameters (i.e., $\partial \mathbf{y} / \partial \boldsymbol{\theta} = \mathbf{0}$),
6. the filter implementation uses intrinsic matrix/vector operators with no provision for sparse matrices, so the subroutines should only be used for small problems.

The output should be directed to file "tstf.lis". This code is used for Example 11.1.

### 13.9.2 Program MLE_PARAMETERID

Program MLE_PARAMETERID is a general-purpose driver for computing the maximum-likelihood estimates of model parameters. It uses a Kalman filter to compute partial derivatives of the filter measurement innovations (residuals) and covariance with-respect-to (WRT) the model parameters. This is used to compute the negative log likelihood, gradient WRT model parameters, and Fisher information matrix. These variables are used in constrained scoring (Gauss-Newton) iterations to converge on the model parameters minimizing the negative log likelihood. Typical parameters that can be estimated via MLE include 1) initial state, 2) measurement noise variances, 3) process noise PSD, and 4) dynamic constants (e.g., feedback constants of Markov processes).

MLE_PARAMETERID is the driver for the filter calculations, and it calls module MODELMOD to obtain the dynamic and measurement model information. Two versions of MODELMOD.F90 are included: a 4-state tracking problem used in program TST_FIS2, and a 6-state model that is identical to the 4-state but also includes measurement biases. A separate program, SIMMEAS.F90 generates simulated range and bearing measurements for the 4-state problem.

The selection of parameters to be estimated is controlled by a file listing the status and prior information for filter states and parameters. The "keyword-type" input text file sets control parameters, initial state values and defines parameters to be estimated via MLE. The acceptable keyword options are:

1) MeasFile: defines the input file name of measurements,
2) TEpoch: sets the start time of measurement processing,
3) Tend: sets the end time of measurement processing,
4) Rvar: defines the measurement noise variance(s) for a given sensor (first input #),
5) Bayesian: if the keyword is present, the MLE solution will be Bayesian (prior estimates are weighted in solution),
6) Debug: sets debug print flags for up to 10 different options (only indices 1, 4, 5, and 6 are currently used),
7) optlSet: sets the maximum number of scoring iterations and the convergence test,

8) LMopt: indicates that Levinburg-Marquard is to be used rather than simple scoring steps, and sets the initial and maximum allowed lambda values,
9) StateInit: denotes that the following records will input state initial values:
   index = state index
   name = state name as hard-coded in modelMod
   x0 = initial state value at TEpoch
   sig0 = 1-sigma uncertainty
   Qs = power spectral density of process noise
   const = constant associated with the state (e.g, Markov process feedback constant)
10) endState: terminates StateInit input,
11) xParm: denotes that the following records will input MLE parameter values:
   index = parameter index
   name = parameter name: The names **must** be of the following form:
      C-xx is a model constant where the number (C-1, C-2, C-3, ...) is defined in modelMod
      QS-xx is a model process noise power spectral density constant where the number (QS-1, QS-2, QS-3, ...) is the state index (as defined in modelMod)
      R-xy is a model measurement noise variance where the first number (x) is the sensor index, and the 2nd number (y) is the index of the measurement for the sensor, e.g., R-12 is for sensor 1, measurement type 2
      x0-xx is the initial value for state xx, i.e., x0-1, x0-2, ...
   xparm = initial estimate of parameter (will ovveride values from above)
   sig0 = 1-sigma uncertainty on parameter if a Bayesian solution is specified
12) endXparm: terminates xParm input.

### 13.9.3 Program ACTRK_FIL_JUMP

Program ACTRK_FIL_JUMP is an EKF filter that tracks a simulated aircraft using the jump detection/estimation algorithm described in Section 11.3.4 for detecting and adapting to maneuvers. It uses the eight-state aircraft model described in Section 3.2.2. Aircraft maneuvers are defined in input file “mnvs.dat”. Output plot files “ACTrk.plt” lists the true (simulated) states versus time, “Filt_Xerr.plt” lists the filter state estimate errors and “Filt_sig.plt” lists the $1-\sigma$ uncertainty. Module JUMPCOM contains all jump detection/estimation algorithms, and must be linked with ACTRK_FIL_JUMP. This is used in Example 11.4.

### 13.9.4 Program ADAPTFILT_TANK

Program ADAPTFILT_TANK is an adaptive multiple-model tank tracking filter. The five different models are defined in Section 3.2.1. The optimal prediction during the projectile time-of-flight is obtained by integrating the state vector from the filter with the greatest likelihood function. The measurement data are obtained from the reconstructed ATMT Scout maneuver data in input file “scoutSim4-xo.plt”. The Kalman filter is implemented in single precision using a U-D filter because a prototype operational implementation is implemented in single precision. The output report should be directed to a file; for example, “afilts.lis”. Program ADAPTFILT_TANK is used in Example 11.5.

## *13.10 Chapter 12: Empirical Modeling*

### 13.10.1 Program TST_CVA3

Program TST_CVA3 tests different approaches for spectral estimation and ARMA modeling. The six approaches are (1) periodogram, (2) BT correlogram, (3) YW MEM, (4) Burg's MEM, (5) Marple MEM, and (6) CVA. Simulated measurements are generated using a fourth-order ARMA model consisting of two loosely damped second-order Markov processes where measurements $y_1(t_i)$ are a function of only one second-order section and the $y_2(t_i)$ measurements are a function of both sections. A control signal of $u(t_i) = 0.8\sin(2\pi t_i / 25)$ feeds the second second-order section and $y_2(t_i)$. The undamped natural frequencies of the two sections are 0.1 and 0.2 Hz. A total of 500 measurements at a sampling interval of 1.0 seconds are generated. The output plot files are

1. "ARMAdata.plt" contains the simulated measurements and control versus time,
2. "PSD_fft.plt" contains the power spectrum computed by the FFT periodogram,
3. "PSD_BT.plt" contains the power spectrum computed by the BT correlogram,
4. "PSD_YW.plt" contains the power spectrum computed by the YW method,
5. "PSD_mem.plt" contains the power spectrum computed by Burg's MEM,
6. "ypsd.plt" contains the true power spectrum of the simulated data,
7. "ypsd_cva.plt" contains the power spectrum computed by the CVA state-space model
8. "ypsd_corr1.plt" contains the power spectrum computed from the correlation matrices generated by CVA.

TST_CVA3 output is used in Examples 12.1, 12.2, and 12.3. Program TST_CVA3 links in subroutines CVA, MEMOPT, SPECT, BT_SPECT, and MARPLE_SPECT from the library. Note that these modules are general-purpose and may be used for alternate models.