Implementing Modified Burg Algorithms in Multivariate Subset Autoregressive Modeling

A. Alexandre Trindade*

February 3, 2003

Abstract

The large number of parameters in subset vector autoregressive models often leads one to procure fast, simple, and efficient alternatives or precursors to maximum likelihood estimation. We present the solution of the multivariate subset Yule-Walker equations as one such alternative. In recent work, Brockwell, Dahlhaus, and Trindade (2002), show that the Yule-Walker estimators can actually be obtained as a special case of a general recursive Burg-type algorithm. We illustrate the structure of this Algorithm, and discuss its implementation in a high-level programming language. Applications of the Algorithm in univariate and bivariate modeling are showcased in examples. Univariate and bivariate versions of the Algorithm written in Fortran 90 are included in the appendix, and their use illustrated.

Keywords: binary tree, Fortran 90, pointer linked list, recursive algorithm, Yule-Walker estimation

1 Introduction

AutoRegressive Moving Average (ARMA) models are well-known and popular in the time series literature. Among others, they are extensively used to model economic and electrical systems whose evolution in time (modulo preliminary transformations, de-trending, and de-seasonalizing) can be well approximated by that of a stationary process. Stationarity constrains a process to have second order properties that do not evolve with time, i.e.

*Department of Statistics, University of Florida, P.O. Box 118545, Gainesville, FL 32611-8545 (trindade@stat.ufl.edu)
constant mean, and a covariance structure between any two observations that depends only on the distance in time (the lag) separating them.

In the multivariate setting, one attempts to model the joint behavior of several univariate series over the same span of time. The usefulness of multivariate ARMA models here though, is stymied by identifiability issues concerning the model parameters (see for example Brockwell and Davis 1991, sec. 11.5). A common alternative is to restrict attention to two special cases: invertible Moving Average (MA) models, and causal AutoRegressive (AR) models, whose parameters are uniquely determined by the second order properties of the process. AR models (called Vector Autoregressive (VAR) in the multivariate case) are often favored over MA models due to their interpretability, simplicity of estimation, and ease of forecasting. They are extensively used in signal processing for modeling various phenomena associated with speech and audio; see for example Godsill and Rayner (1998).

The d-dimensional vector process \( \{X_t, \quad t = 0, \pm 1, \ldots\} \), is said to be a VAR process of order \( p \), \( \text{VAR}(p) \), if it is a stationary solution of the equations,

\[
X_t = \Phi(1)X_{t-1} + \cdots + \Phi(p)X_{t-p} + Z_t,
\]

where, \( \Phi(1), \ldots, \Phi(p) \), are \((d \times d)\) constant matrices (the VAR coefficient matrices), and \( \{Z_t\} \) is a sequence of zero-mean uncorrelated random vectors, each with covariance matrix \( \Sigma \). We call the process \( \{Z_t\} \) white noise, and write \( \{Z_t\} \sim \text{WN}(0, \Sigma) \). The autocovariance function of \( X_t \) is,

\[
E[X_{t+h}X'_t] = \Gamma(h), \quad h = 0, \pm 1, \ldots
\]

A \( \text{VAR}(p) \) is therefore a linear regression of the current value of the series on its previous \( p \) values i.e. an autoregression. We say that we are modeling the series on the lagged set \( \{1, \ldots, p\} \).

One can generalize this to modeling on a lagged subset \( K = \{k_1, \ldots, k_m\} \subseteq \{1, \ldots, p\} \), with \( k_1 < \cdots < k_m \equiv p \), and the coefficient matrices pertaining to the lags not present in the set \( K \), constrained to be zero. Such models are called Subset Vector Autoregressive (SVAR; SAR in the univariate case), and take the form

\[
X_t = \Phi_K(k_1)X_{t-k_1} + \cdots + \Phi_K(k_m)X_{t-k_m} + Z_t, \quad \{Z_t\} \sim \text{WN}(0, U_K). \quad (1)
\]

SVAR models are appropriate in situations where one does not wish to include all the lags of the complete (full-set) VAR model. Two such instances are:
Modeling of seasonal time series. If \( B \) denotes the backward shift operator, i.e. \( B^k X_t = X_{t-k} \) for any positive integer \( k \), then causal SAR models of the form,

\[
(1 - \psi B^s)(1 - \phi_1 B - \cdots - \phi_p B^p) X_t = Z_t,
\]

will exhibit approximate cyclical behavior for appropriate values of the coefficients \( \psi, \phi_1, \ldots, \phi_p \), and orders \( s, p \), as evidenced by sharp peaks in the spectral density. This suggests that some seasonal time series can effectively be modeled as SAR processes.

Figure 1 shows a realization from the SAR(3),

\[
X_t - 0.99X_{t-3} = Z_t, \quad \{Z_t\} \sim \text{WN}(0, 1),
\]

along with a plot of the spectral density function of the process on the interval \((0, \pi)\). The spectral density peaks at a frequency of \(2\pi/3\) radians per unit time, which corresponds to a period of length 3.

Fitting best subset models. As in linear regression, one can search for the “best” subset AR/VAR model up to some maximum order, \( p \). “Best” can be measured by one’s favorite information criterion, such as Akaike (AIC), Bayesian (BIC), Schwarz (SIC), or even Minimum Description Length (MDL). Researchers have devised efficient algorithms to perform this search. One of the earliest attempts was made by McClave (1975), who used an algorithm adapted from linear regression. Penn and Terrell (1982), introduced an algorithm recursive in the maximum lag for best subset identification. Zhang and Terrell (1997) refine the search by inspecting certain statistics. Rather than performing an exhaustive search through all \( 2^p \) models, Sarkar and Sharma (1997) propose a statistical method for identifying the best subset.

Figure 2 shows the celebrated Canadian Lynx Trappings data. Ecological oscillations in predator-prey populations, mean that the logarithms of this data set are often modeled as a SAR process; a perennial favorite in the SAR literature. The lower part of the figure shows a spectral density estimator (the periodogram) for this data, which suggests the period of the oscillations to be approximately \(2\pi/0.6 \approx 10.5\) years. In section 4, we will apply the algorithm of section 2 to perform an exhaustive search for the best SAR model.

For a given SVAR model order, one typically wishes to find maximum likelihood (ML) estimates of the parameters. Using standard arguments,
Figure 1: The process \( X_t - 0.99X_{t-3} = Z_t \), \( \{Z_t\} \sim \text{WN}(0,1) \). Top: a realization of the process with Gaussian noise. Bottom: the corresponding spectral density function.

The \(-2\) log likelihood for the vectors \( \mathbf{X}_1, \ldots, \mathbf{X}_n \) from the Gaussian SVAR process of dimension \( d \) defined by equation (1), can be shown to be

\[
\mathcal{L} (\Phi_K(k_1), \ldots, \Phi_K(k_m), U_K) = nd \log(2\pi) + \log \det(\Gamma_{km}) \\
+ (n - k_m) \log \det(U_K) + [\mathbf{X}'_1, \ldots, \mathbf{X}'_{k_m}], \mathbf{X}'_1, \ldots, \mathbf{X}'_{k_m}]' \\
+ \sum_{t=k_m+1}^{n} \left[ \mathbf{X}_t - \sum_{j \in K} \Phi_K(j) \mathbf{X}_{t-j} \right]' U_K^{-1} \left[ \mathbf{X}_t - \sum_{j \in K} \Phi_K(j) \mathbf{X}_{t-j} \right], \tag{2}
\]

where \( \Gamma_{km} = \mathbb{E} \left( [\mathbf{X}'_1, \ldots, \mathbf{X}'_{k_m}][\mathbf{X}'_1, \ldots, \mathbf{X}'_{k_m}] \right) \).

**Remark 1**
The potentially large number of parameters involved in ML estimation
(d^2k_m + \frac{d^2+d}{2} of them), and the possible existence of many local minima which are much larger than the global minimum, makes the numerical search for the minimizers a difficult problem. The feasibility of ML estimation is therefore highly dependent upon good initial estimates.

For this, and the reason that one may wish to avoid ML estimation altogether, it is important to consider alternative fast and simple SVAR estimation methods for obtaining models with high likelihoods. Recently, Brockwell, Dahlhaus, and Trindade (2002), introduced a method for doing just that. Their method, the BDT Algorithm which we consider in section 2, is recursive in the model order, parameter estimates of larger order models being constructed from those of smaller order models. Since the Brockwell et al. (2002) paper focuses mostly on theoretical aspects, the main purpose of this article is to serve as a pragmatic complement to it in the following
ways:

(i) Elucidate the recursive structure of the Algorithm.

(ii) Discuss the main issues involved in implementing the Algorithm in a high-level programming language like Fortran 90.

(iii) Provide coded versions of the Algorithm along with examples that illustrate its usage.

Section 3 accomplishes the first two goals, in the framework of a binary tree of pointer-linked nodes. The examples are presented in Section 4, which also illustrates some meta applications of the Algorithm. (Accompanying data sets are provided in Appendix B.) Fortran 90 programs implementing a univariate (BDT.F90) and a bivariate (BDT2.F90) version of the Algorithm are provided in Appendix C. Appendix A summarizes the function of the principal subroutines in these programs.

2 Estimation Methods

In this section we discuss alternative SVAR parameter estimation methods to ML. The first is a generalization of the well-known Yule-Walker method of moments estimator for full-set modeling, and has not previously appeared in the literature in this form. The second is a flexible recursive Burg-type algorithm, introduced by Brockwell et al. (2002), whose structure and implementation is the main focus of this paper. In order to introduce both methods, we will need to consider not only the (forward) SVAR model (1), but also the backward SVAR model

\[ X_t = \sum_{j \in K^*} \Psi_{K^*}(j) X_{t+j} + Z_t, \quad \{Z_t\} \sim \text{WN}(0, V_{K^*}), \]

where \( K^* = \{k_m - k_{m-1}, \ldots, k_m - k_1, k_m\} \), and suppose that the process \( \{X_t\} \) is causal (meaning that the current value of the series can be expressed as a function of current and past values of the white noise sequence as, \( X_t = \sum_{j=0}^{\infty} Y_j Z_{t-j} \)).

2.1 Non-Recursive Estimation: The Yule-Walker Equations

If we multiply both sides of (1) by \( X_{t-i} \), \( i = 0, k_1, \ldots, k_m \) in turn, and (noting the causal representation) take expectations, we obtain the so-called
Yule-Walker (YW) equations:
\[ \Gamma(k) = \sum_{j \in K} \Phi_K(j) \Gamma(k-j), \quad k \in K, \tag{4} \]
\[ U_K = \Gamma(0) - \sum_{j \in K} \Phi_K(j) \Gamma(j)'. \tag{5} \]

For the backward SVAR model (3), the corresponding YW equations are
\[ \Gamma(k)' = \sum_{j \in K^*} \Psi_K^*(j) \Gamma(j-k), \quad k \in K^*, \tag{6} \]
\[ V_{K^*} = \Gamma(0) - \sum_{j \in K^*} \Psi_K^*(j) \Gamma(j). \tag{7} \]

Now define \( R_K \) and \( G_K \) to be matrices of autocovariances as follows: with \( k_0 = 0 \), define the \((i,j)\)th, \( i,j = 1, \ldots, m + 1 \), block-matrix entry of \( R_K \) to be,
\[ (R_K)_{i,j} = \begin{cases} 
\Gamma(k_{j-1} - k_{i-1}), & j \geq i \\
\Gamma(k_{j-1} - k_{i-1})', & j < i 
\end{cases} \]
and \( G_K \) obtained from \( R_K \) by striking out the first block row and column. The \( m + 1 \) forward YW equations can now be succinctly written in block-matrix form as,
\[ [I_d, -\Phi_K(k_1), -\Phi_K(k_2), \ldots, -\Phi_K(k_{m-1}), -\Phi_K(k_m)] R_K = [U_K, 0, \ldots, 0], \]
and the backward YW equations as,
\[ [-\Psi_K^*(k_m), -\Psi_K^*(k_{m-1} - k_1), \ldots, -\Psi_K^*(k_{m} - k_{m-1}), I_d] R_K = [0, \ldots, 0, V_{K^*}]. \]

Defining
\[ \Gamma_K \equiv [\Gamma(k_1), \Gamma(k_2), \ldots, \Gamma(k_{m-1}), \Gamma(k_m)], \]
and \[ \Phi_K \equiv [\Phi_K(k_1), \Phi_K(k_2), \ldots, \Phi_K(k_{m-1}), \Phi_K(k_m)], \]
we can write (4)-(5) in the reduced block-matrix form
\[ \Gamma_K = \Phi_K G_K, \tag{8} \]
\[ U_K = \Gamma(0) - \Phi_K \Gamma_K'. \tag{9} \]

These can now be solved for \( \Phi_K \) and \( U_K \):
\[ \Phi_K = \Gamma_K G_K^{-1}, \tag{10} \]
\[ U_K = \Gamma(0) - \Gamma_K G_K^{-1} \Gamma_K'. \tag{11} \]
where $G^{-1}_K$ denotes any generalized inverse of $G_K$. The solution $\Phi_K$, gives the minimum mean-squared error linear predictor of $X_t$ in terms of $X_{t-i}$, $i \in K$. Its mean-squared error is $U_K$. Analogous results hold for the backward YW equations.

When fitting SVAR model (1) to a set of observations $x_1, \ldots, x_n$ from the zero-mean random vectors $X_1, \ldots, X_n$, one of the simplest approaches is to substitute sample estimates for the autocovariances in (10) and (11).

Taking the usual estimator of the autocovariance matrix at lag $h$ to be,

$$\hat{\Gamma}(h) = \begin{cases} \frac{1}{n} \sum_{t=1}^{n-h} x_{t+h}x_t', & \text{if } h \geq 0, \\ \hat{\Gamma}(-h)', & \text{if } h < 0, \end{cases}$$

the resulting method of moments estimates are,

$$\hat{\Phi}_K = \hat{\Gamma}_K \hat{G}^{-1}_K; \quad \hat{U}_K = \hat{\Gamma}(0) - \hat{\Phi}_K \hat{\Gamma}'_K.$$  

These are the so-called YW estimates in the full-set case, and we will refer to their subset generalization by the same name. The fitted YW SVAR model is therefore,

$$X_t = \hat{\Phi}_K(k_1)X_{t-k_1} + \cdots + \hat{\Phi}_K(k_m)X_{t-k_m} + \mathbf{Z}_t, \quad \{\mathbf{Z}_t\} \sim \text{WN}(0, \hat{U}_K). \quad (15)$$

### 2.2 Recursive Estimation: The BDT Algorithm

By defining the empirical forward and backward prediction error residuals $\hat{\varepsilon}_K$ and $\hat{\eta}_{K^*}$, associated with models (1) and (3) as,

$$\hat{\varepsilon}_K(t) = x_t - \sum_{i \in K} \hat{\Phi}_K(i)x_{t-i}, \quad \text{and} \quad \hat{\eta}_{K^*}(t) = x_t - \sum_{j \in K^*} \hat{\Psi}_{K^*}(j)x_{t+j},$$

Brockwell et al. (2002), introduce a family of SVAR model parameter estimators, based on Burg’s (1968) recursive algorithm. Their BDT Algorithm, takes the following form.
Algorithm 1 (The BDT Algorithm)

\begin{align*}
\hat{\Phi}_K(k_m) &= \ldots \\
\hat{\Phi}_K(i) &= \hat{\Phi}_J(i) - \hat{\Phi}_K(k_m)\hat{\Psi}_J(k_m-i), \quad i \in J \\
\hat{\Psi}_{K^*}(k_m) &= \hat{\Psi}_J^* \hat{\Phi}_K(k_m) \hat{\Psi}_J \hat{V}_J^{-1} \\
\hat{\Psi}_{K^*}(j) &= \hat{\Psi}_J^*(j) - \hat{\Psi}_{K^*}(k_m)\hat{\Phi}_{J^*}(k_m-j), \quad j \in J^* \\
\hat{U}_K &= \hat{U}_J - \hat{\Phi}_K(k_m)\hat{V}_J \hat{\Phi}_K(k_m)' \\
\hat{V}_{K^*} &= \hat{V}_J^* - \hat{\Psi}_{K^*}(k_m)\hat{U}_J \hat{\Psi}_{K^*}(k_m)' \\
\hat{\epsilon}_K(t) &= \hat{\epsilon}_J(t) - \hat{\Phi}_K(k_m)\hat{\eta}_{J^*}(t-k_m) \\
\hat{\eta}_{K^*}(t) &= \hat{\eta}_{J^*}(t) - \hat{\Psi}_{K^*}(k_m)\hat{\epsilon}_J(t+k_m)
\end{align*}

with initial conditions,

\begin{align*}
\hat{\epsilon}_\emptyset(t) &= \hat{\eta}_\emptyset(t) = \begin{cases} x_t, & t \in \{1, \ldots, n\}, \\
0, & \text{otherwise,}
\end{cases} \\
\hat{U}_\emptyset &= \hat{\Gamma}(0) = \hat{V}_\emptyset,
\end{align*}

and the sets \( J \) and \( J^* \), formed from the sets \( K \) and \( K^* \), respectively, by omitting \( k_m \).

A variety of different estimators can be obtained by an appropriate selection of the boxed reflection coefficient expression in (16). Brockwell et al. (2002), note that the choice

\begin{align*}
\hat{\Phi}_K(k_m) = \left( \frac{1}{n} \sum_{t=1}^{n+k_m} \hat{\epsilon}_J(t)\hat{\eta}_{J^*}(t-k_m)' \right) \hat{V}_J^{-1}
\end{align*}

(22)

gives precisely the YW estimators (13) and (14) (reformulated via similar recursions, the resulting Algorithm is known as Levinson-Durbin), but that selecting \( \hat{\Phi}_K(k_m) \) to be the minimizer of the weighted sum of forward and backward prediction errors

\begin{align*}
\sum_{t=k_m+1}^{n} \left[ \hat{\epsilon}_K(t)'A\hat{\epsilon}_K(t) + \hat{\eta}_{K^*}(t-k_m)'B\hat{\eta}_{K^*}(t-k_m) \right],
\end{align*}

(23)

tends to produce models with consistently higher Gaussian likelihoods. By selecting different weight matrices \( A \) and \( B \), they propose a total of three additional methods: Burg, Vieira-Morf, and Nuttall-Strand, each being a
plausible subset generalization of existing full-set analogues with the same name.

The BDT Algorithm necessarily couples together the forward and backward modeling problems. Arranging the elements of $K$ on the number line as shown in Figure 3, allows us to better visualize this coupling. The forward set of lags, $K$, are simply the distances of the elements of $K$ from the origin; while the backward set of lags, $K^*$, are the corresponding distances from $k_m$.

Note that the YW estimator, $\hat{\Phi}_K(k_m)$, obtained from (13), requires the inversion of $\hat{G}_K$, which is of dimension $md$. Recursive algorithms are better suited to searching for a best subset model with a specified maximum number of lags, and involve inversion of matrices whose dimension is at most $d$ ($d^2$ in some instances).

3 Implementing the BDT Algorithm

Apart from the special case of YW, estimators arising from the BDT Algorithm cannot in general be reformulated in a non-recursive manner. The intricate structure of the recursions, a by-product of the forward and backward model coupling, can seem rather daunting from a programming perspective. In this section therefore, we discuss the main issues involved in implementing this Algorithm in a high-level programming language like Fortran 90. An important goal is to minimize computing time, and our approach will be to create a linked list of nodes in the form of a binary tree. In the process, we will make use of recursive pointers, recursive subroutines, and data types that incorporate recursive definitions.

3.1 Building a binary tree of linked nodes

The recursive solution of the equations defining the BDT Algorithm, generates a collection of estimators of SVAR models of increasing orders, until the required order is reached. Suppose modeling on the set of lags $\{1, 3, 7\}$
is desired. To determine where application of the algorithm should begin, we first need to work down to derived subsets of lags comprised of just one lag. This is done by successively forming the $J$ and $J^*$ subsets of lags for each parent set of lags $K$, as shown in Figure 4. Each of the subsets $J$ and $J^*$ then assumes the role of a parent lag, $K$, and the procedure is repeated. In the resulting binary tree structure, we will refer to all the modeling information pertaining to a set of lags as a node. The number of lags in a node will define its level in the tree. The strategy for this recursive tree-building will then be as follows:

**Pseudocode 1 (Build Tree)**

```plaintext
level := m
while level > 1 do
    for each node in current level do
        compute lags in $J$ and $J^*$ subnodes
        direct pointers to $J$ and $J^*$ subnodes
        level := level - 1
    od
level := level - 1
od
```

Figure 4: Binary tree of linked nodes for modeling on the set of lags $\{1, 3, 7\}$. 

$$
K = \{1, 3, 7\} \\
J \quad J^* \\
K = \{1, 3\} \\
J \quad J^* \\
K = \{4, 6\} \\
J^* \quad J \\
K = \{1\} \\
J^* \\
K = \{2\} \\
J \\
K = \{4\}
$$
For programming, it will be necessary to rewrite the backward model equations (17), (18), (19), and (21), in the unstarred lags format:

\[
\hat{\Psi}_K(k_m) = \hat{V}_J \hat{\Phi}_{K*}^t(k_m) \hat{U}_J^{-1} ,
\]
\[
\hat{\Psi}_K(j) = \hat{\Psi}_J(j) - \hat{\Psi}_K(k_m) \hat{\Phi}_{J^*}(k_m - j), \quad j \in J,
\]
\[
\hat{V}_K = \hat{V}_J - \hat{\Psi}_K(k_m) \hat{U}_J \hat{\Phi}_K(k_m)^t ,
\]
\[
\hat{\eta}_K(t) = \hat{\eta}_J(t) - \hat{\Psi}_K(k_m) \hat{\varepsilon}_{J^*}(t + k_m) ,
\]

obtained by noting that \((K^*)^* = K\), and \((J^*)^* = J\). The unstarred lags representation carifies how the backward model estimates should be computed for any given node. Letting \texttt{node} be a user-defined derived data type, and \(K = \{k_1, \ldots, k_m\}\) denote a generic set of lags for a given node, this data type should then consist of the following components:

(i) \texttt{lags} - vector containing the current set of lags, \(K\), on which modeling is desired (type \texttt{integer}).

(ii) \texttt{level} - scalar specifying the level of the node in the tree (type \texttt{integer}).

(iii) \(\Phi_K\), \(\Psi_K\) - vectors of forward and backward model coefficient matrices; that is \(\Phi_K = \{\Phi_K(k_1), \ldots, \Phi_K(k_m)\}\), and \(\Psi_K = \{\Psi_K(k_1), \ldots, \Psi_K(k_m)\}\) (type \texttt{real}).

(iv) \(U_K\), \(V_K\) - estimates of the white noise covariance matrices for the forward and backward modeling problems (type \texttt{real}).

(v) \(\varepsilon_K(t)\), \(\eta_K(t)\) - vectors of prediction error residuals for the forward and backward modeling problems (type \texttt{real}).

(vi) \texttt{reg}, \texttt{str} - pointers to the \(J\) and \(J^*\) subnodes, respectively, one level below the current level (type \texttt{node}, defined recursively).

The tree can be linked by recursive calls to a \texttt{RECURSIVE SUBROUTINE}, with \texttt{level}, \texttt{lags}, and a \texttt{pointer} of type \texttt{node} as arguments. This routine should also set a flag to signal when a particular node has been initialized (linked in the list), but the remaining components, (iii)-(vi), not yet evaluated (node \texttt{unfilled}). The flag, setting the first row and column entry of \(U_K\) to zero for example, will be used by a subsequent node-filling routine. Two pointers should emanate from each node, \texttt{reg} pointing to \(J\), and \texttt{str} to \(J^*\). At level 1, these pointers should point nowhere (\texttt{NULLIFY}). From Figure 4, we note that both nodes at level 2 have the set \(\{2\}\) as their \(J^*\) subnode. Two copies of this subnode can be made, each linked to its appropriate parent node. This duplication of nodes can be avoided by a more complex program, since otherwise exactly \(2^m\) nodes are created for modeling on \(m\) initial lags.
3.2 Filling the nodes

Once the tree with all appropriate linking pointers is in place, we will need to evaluate the remaining components, (iii)-(vi), of each node. This can be done by “walking” through the tree, following the linked list of nodes. Once again, a RECURSIVE SUBROUTINE taking a pointer as argument can be employed to achieve this, certifying first that each node has not yet been filled by checking the flag alluded to earlier.

The recursion should be implemented in such a way that the tree is walked to level 1 where the node-filling can begin. From Figure 4, we note that this involves filling nodes \{1\}, \{2\}, and \{4\} first. With these, we can now fill nodes \{1,3\} and \{4,6\}, at level 2. The operation terminates at the top node, \{1,3,7\}, if the pointer to this node is passed as the original argument to the recursive node-filling subroutine. The pseudo-code for this phase of the implementation could therefore be:

Pseudocode 2 (Fill Tree)

call node filling routine with pointer to top node as argument
while level of current node > 1 do
    call node filling routine with reg pointer as argument
    if current node unfilled then fill it fi
    call node filling routine with str pointer as argument
    if current node unfilled then fill it fi od
if level of current node = m then fill top node od

This coding will give filling-precedence to nodes at low levels that emanate from parents with respect to which they are \(J\) subnodes. In example 4, this would result in the following filling order: \{1\}, \{2\} (\(J^*\) subnode of \{1,3\}), \{1,3\}, \{4\}, \{2\} (\(J^*\) subnode of \{4,6\}), \{4,6\}, \{1,3,7\}.

Note that in the univariate case there is no distinction between forward and backward model parameters for the same set of lags; that is, the YW equations give \(\hat{U}_K \equiv \hat{V}_K\), and \(\hat{\Phi}_K \equiv \hat{\Psi}_K\), for any set \(K\). Both this and the fact that all parameters are scalars, greatly simplifies the programming task when \(d = 1\).

4 Examples

Included in Appendix C are BDT.F90 and BDT2.F90. These are, respectively, univariate and bivariate Fortran 90 programs implementing the BDT
Algorithm. The programs utilize a few linear algebra subroutines in the International Mathematical and Statistical Library (IMSL). In this section we document how to run the programs in order to fit a particular SAR/SVAR model to a given data set, and illustrate some potential meta applications that involve repeated modeling with each of the programs in the inner loop. (We have not provided the programs for Examples 3-5, but they are available from the author upon request.)

4.1 Example 1: Running BDT.F90

In order to fit the model

$$X_t = \phi_K(1)X_{t-1} + \phi_K(2)X_{t-2} + \phi_K(3)X_{t-3} + \phi_K(4)X_{t-4} + \phi_K(10)X_{t-10} + \phi_K(11)X_{t-11} + Z_t, \quad \{Z_t\} \sim WN(0, \sigma^2),$$

of Table 2 to the mean-corrected base 10 logarithms lynx data (lynx10.tsm in Appendix B) of Figure 2, we ran the compiled version of BDT.F90 with the following inputs at the prompt (>):

```
File name of time series for modelling: lynx10.tsm
Do you wish to mean-correct the observations (1=yes, 0=no)? 1
There are 114 observations.
First obs is -0.47391147326671135 last is 0.6273039283027888
Enter the number of lags to be modeled (<27): 6
Enter the lags: 1 2 3 4 10 11
Enter the method for obtaining the reflection coefficients: Yule-Walker (1), Burg (2), Vieira-Morf (3), Nuttall-Strand (4):

The estimated subset Burg AR coefficients are:
Phi( 1): 1.15639
Phi( 2): -0.50191
Phi( 3): 0.19869
Phi( 4): -0.21127
Phi(10): 0.37899
```
Phi(11): -0.42454
**************************************************
Burg WN variance estimate : 3.61762021546651741E-2
RSS/n WN variance estimate: 3.69130827522938937E-2
-2 Log Likelihood (Burg) : -46.962408999128101
-2 Log Likelihood (RSS/n): -46.985742242956121
AICC (RSS/n) : -31.929138811254461
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
This gives the Burg estimators. The Yule-Walker estimates can be obtained by selecting “1” in the last step. The remaining estimators proposed by Brockwell et al. (2002), can be obtained by selecting “3” (Vieira-Morf), and “4” (Nuttall-Strand). The RSS/n is the MLE of $\sigma^2$ for the given set of SAR coefficient estimates, $\{\hat{\phi}_K(k_1), \ldots, \hat{\phi}_K(k_m)\}$, thus it comes as no surprise that the -2 Log Likelihood based on it is never larger than the -2 Log Likelihood based on the Burg $\hat{\sigma}^2$. The AICC is a bias-corrected version of AIC; see Brockwell and Davis (1991).

4.2 Example 2: Running BDT2.F90

To fit the bivariate SVAR model

$$X_t = \Phi_K(1)X_{t-1} + \Phi_K(3)X_{t-3} + Z_t, \quad \{Z_t\} \sim WN(0, \Sigma_K),$$

to the sun2.tsm data (Appendix B), we ran the compiled version of BDT2.F90 with the following inputs at the prompt (>):

Enter file name of time series for modelling:
>sun2.tsm
Do you wish to mean-correct the observations (1=yes, 0=no)?
>1
There are 50 observations.
First obs is 53.5200 last is 35.6200
First obs is 10.4800 last is 27.6200
Enter the number of lags to be modeled (<27):
>2
Enter the lags:
>1 3
Is the true white noise covariance matrix known (1=yes, 0=no)?
Enter the method for obtaining the reflection coefficients.
Yule-Walker (1), Burg (2), Vieira-Morf (3), Nuttal-Strand (4):

Estimated subset Morf coefficient matrices:
Phi(1):
-0.853995 1.571658
-0.913452 1.279817
Phi(3):
0.029511 0.092263
0.291517 -0.150232

Estimated Morf (forward) WN covariance matrix:
145.678543 220.305063
220.305063 580.954041

-2 Log Like (Morf) : 812.877439308433
-2 Log Like (RSS/n): 812.820447412800

Estimated RSS/n (forward) WN covariance matrix:
143.844793 221.765063
221.765063 598.346541

Like the univariate program, two estimates of $U_K$ are given: the first is method-specific (method 3, Vieira-Morf, in this case), the second, RSS/n, is the MLE of $U_K$ holding $\Phi_K(1)$ and $\Phi_K(3)$ fixed at their estimated values. In the multivariate case, there is no known closed form for the RSS/n estimate like there was in the univariate case (see Appendix A.6). If the data was obtained via simulation and a “1” was entered at the 5th prompt, the program would use the furnished value of $U_K$ as the initial guess for RSS/n in the optimizing routine. Since we entered “0”, the program will use the Morf WN estimate as the initial guess.

4.3 Example 3: Best Subset Searching

The lynx data of Figure 2 is often cited in the literature in connection with SAR modeling. Using YW estimation and their own respective non-exhaustive search algorithms, Tong (1977), Penm and Terrell (1982), Zhang
and Terrell (1997), and others, identify a SAR(1,2,4,10,11), i.e. \( K = \{1, 2, 4, 10, 11\} \), as the best SAR model according to a variety of information criteria. It is important to realize that some of these search methods are non-exhaustive and statistical in nature, and will therefore not guarantee a correct identification with certainty.

Using the BDT Algorithm, we performed an exhaustive search for the minimum AICC SAR model, for the mean-corrected base 10 logarithms of the lynx data. Letting \( p \) denote the maximum lag considered in the search (meaning that \( 2^p \) models had to be checked), we considered \( p = 4 \), 8, and 12, in turn. This set of searches was performed for each of the YW and Burg estimation methods. The number of subsets out of the \( 2^p \) that resulted in non-causal fitted models, as well as the corresponding computational (CPU) times taken by each search, were recorded. The AICC (RSS/n) of the best SAR model was also computed.

Table 1: Results of best (minimum AICC) SAR model search for the mean-corrected base 10 logarithms of the Canadian lynx data of Figure 2.

| Estimation method | \( p \) | Lags in best subset | AICC of best model | Prop. of non-causal models | CPU time (secs) |
|------------------|------|--------------------|-------------------|---------------------------|----------------|
|                  | 4    | 1,2,4              | -9.89             | 3/16                      | 1.2            |
| YW               | 8    | 1,2,4,8            | -16.17            | 78/256                    | 21.3           |
|                  | 12   | 1,2,4,10,11        | -31.80            | 1392/4096                 | 637.2          |
| Burg             | 4    | 1,2,4              | -10.08            | 3/16                      | 1.8            |
|                  | 8    | 1,2,4,8            | -16.27            | 81/256                    | 29.6           |
|                  | 12   | 1,2,3,4,10,11      | -31.93            | 1489/4096                 | 678.7          |

The results are summarized in Table 1. The best SAR model with maximum lag 12 found by the YW method, coincides with that identified by other researchers as discussed above; but that arrived at by the new subset Burg method, adds lag 3, and has a slightly lower value of AICC. Note also that the proportion of subsets resulting in non-causal fitted models (meaning that a SAR model with these lags is inappropriate for the data), remained steady at approximately 1/3 across all searches. At the same value of \( p \), CPU times for Burg are slightly higher than those for YW, both growing exponentially with \( p \). The computations were carried out on a Sun Enter-
prise 450 unix server, equipped with about 4G of memory. The most severe limiting factor in this type of computation is available memory, since at least \(2^p\) pointers have to be allocated.

In Table 2, we present the parameter estimates of the best SAR model identified by each respective method when \(p = 12\). The constrained ML estimates were obtained via the ITSM2000 package (Brockwell and Davis, 2002), starting with the Burg estimates.

Table 2: Best SAR models fitted to the mean-corrected base 10 logarithms of the Canadian lynx data, as identified by each respective method. The Maximum Likelihood estimates were obtained by starting with the Burg estimates, and constraining the ML search to the same SAR lags.

| Parameter | Estimates by Method |
|-----------|---------------------|
|           | Maximum Likelihood | Burg  | Yule-Walker |
| \(\phi_K(1)\) | 1.148 | 1.156 | 1.094 |
| \(\phi_K(2)\) | -0.502 | -0.502 | -0.357 |
| \(\phi_K(3)\) | 0.199 | 0.199 | |
| \(\phi_K(4)\) | -0.217 | -0.211 | -0.127 |
| \(\phi_K(10)\) | 0.351 | 0.379 | 0.324 |
| \(\phi_K(11)\) | -0.401 | -0.425 | -0.362 |
| \(\sigma^2\) | 0.037 | 0.037 | 0.038 |
| AICC | -32.22 | -31.93 | -31.80 |

4.4 Examples 4 and 5: Adaptive Behavior

In these simulated bivariate examples, we illustrate the component-wise convergence of the Burg and YW estimates to their true values, as a function of observation number or time. This adaptive behavior is important in online applications where it is desirable to monitor the convergence of the estimates, particularly when observation number is still low. Note however that the BDT Algorithm is recursive in the model order, not observation number, and is therefore not truly adaptive in that sense. The entire Algorithm needs to be re-run from scratch whenever a new observation becomes available.

The characteristic polynomial of SVAR model (1) is

\[
P(z) = \det \left[ I_d - \Phi_K(k_1)z^{k_1} - \cdots - \Phi_K(k_m)z^{k_m} \right].
\]
The model is causal if the zeroes of its characteristic polynomial are all greater than one in magnitude. It is well-known that in the univariate full-set case, the YW estimators can be severely biased if the roots of the AR characteristic polynomial are close to the unit circle (quasi-non-stationarity). To allow for the expected dependence of performance on the location of the zeroes of $P(z)$, we considered causal models with different configurations of these zeroes. A total of 250 observations were sequentially simulated from the basic SVAR(2) model,

$$X_t = \Phi X_{t-2} + Z_t \equiv \begin{bmatrix} \Phi_{11} & \Phi_{12} \\ \Phi_{21} & \Phi_{22} \end{bmatrix} X_{t-2} + Z_t, \quad Z_t \sim N_2(0, I_2),$$

and we started estimation at observation number 10.

**Example 4**

$$\Phi = \begin{bmatrix} 0.547 & -0.300 \\ 0.700 & -0.457 \end{bmatrix}, \quad P(z) = (1 - 0.25z^2)(1 + 0.16z^2),$$

with roots of characteristic polynomial: $\pm 2, \pm 2.5i$.

**Example 5**

$$\Phi = \begin{bmatrix} 1.414 & -0.300 \\ 0.700 & 0.497 \end{bmatrix}, \quad P(z) = (1 - 0.98^2z^2)(1 - 0.95^2z^2),$$

with roots of characteristic polynomial: $\pm 1.02, \pm 1.03$.

The results are displayed in Figure 5, where we plot the component-wise departures of the estimated SVAR coefficient matrices from their true values, $\Phi_{ij} - \hat{\Phi}_{ij}$, $i, j = 1, 2$, as a function of observation number. The pattern of convergence between the two methods is similar in Example 4, but dramatically different in Example 5. Although based on a single simulated realization presented only to illustrate a meta application of the bivariate BDT Algorithm, this phenomenon is nevertheless consistent with what has been noted about the behavior of YW versus Burg in quasi-non-stationary modeling. Since both estimators and the MLE all have the same asymptotic distribution, there is little cause for concern with large samples; it is with small samples that one should exercise caution when selecting an estimation method. The more extensive analysis by Brockwell et al. (2002), suggests that Burg is in general a better estimator than YW.
Figure 5: Departures of the 4 estimated components of the SVAR coefficient matrix from their true values, by method, for the simulated realizations of Examples 4 and 5.

5 Conclusion

We have discussed the popularity of multivariate subset autoregressive models, and highlighted the importance of fast, simple, and efficient methods for the estimation of their parameters. One such set of estimators is obtained via the classical Yule-Walker method-of-moments, which we have presented as the solution to a system of simultaneous linear matrix equations. A recently introduced more general estimation method, the BDT Algorithm, is recursive in the order of the fitted model, thus avoiding the (potentially large) matrix inversions required in solving the Yule-Walker equations. By suitably modifying the reflection coefficient calculation, this Algorithm can produce a variety of estimators with different finite sample properties, among them Yule-Walker. We have illustrated the recursive structure of this Algorithm,
and discussed its implementation in a high-level programming language like Fortran 90. The speed of the Algorithm was assessed in finding a best subset model for the Canadian lynx data, and shown, in problems of moderate size, to be a feasible alternative to non-exhaustive search techniques which do not guarantee correct subset identification. We concluded with two simulated bivariate examples that illustrate the adaptive performance of the Yule-Walker and Burg estimators, implemented via the BDT Algorithm. We find that the Burg estimates tend to stabilize more quickly than Yule-Walker, and are far less affected by proximity of the model to non-stationarity.

6 Acknowledgements

The author would like to acknowledge NSF for partial support of this research through grant number DMS-9972015, and the invaluable suggestions provided by the referee who also reviewed the code.

References

[1] Brockwell, P.J., Dahlhaus, R., and Trindade, A.A. (2002), ”Modified Burg Algorithms for Multivariate Subset Autoregression”, Technical Report 2002-015, Department of Statistics, University of Florida.

[2] Brockwell P.J., and Davis R.A. (1991), Time Series: Theory and Methods (2nd ed.), New York: Springer-Verlag.

[3] Brockwell P.J., and Davis R.A. (2002), Introduction to Time Series and Forecasting, Second Ed., New York: Springer-Verlag.

[4] Burg, J.P. (1968), ”A New Analysis Technique for Time Series Data”, in Modern Spectrum Analysis, (1978), D.G. Childers (ed.), NATO Advanced Study Institute of Signal Processing with emphasis on Underwater Acoustics, New York: IEEE Press.

[5] Godsill, S.J., and Rayner P.J.W. (1998), Digital Audio Restoration: A Statistical Model-Based Approach, Berlin: Springer.

[6] Hooke, R., and Jeeves T. (1961), ”A direct search solution of numerical and statistical problems”, Journal of Association for Computing Machinery, 8, 212-229.
A Description of Principal Program Subroutines

As already stated, the core of the subset modeling programs Burg and Burg2 is the globally visible MODULE Tree, with SUBROUTINE Make_Tree its driving subroutine. In this section, we will provide a brief description of the essential functions of its main constituent subroutines.

A.1 Build_Node_Tree

This is a RECURSIVE SUBROUTINE that initializes the tree of nodes by allocating pointers to and from nodes. It takes on the level, lags, and a pointer of type node as arguments. It begins execution at the unique node of level \( m \) (top_node), creating pointers to the \( J \) and \( J^* \) subnodes (this_node%reg and this_node%str, respectively). Following these pointers to level \( m - 1 \), Build_Node_Tree subsequently allocates pointers to the subnodes in level \( m - 2 \). It achieves this by calling itself with the appropriate arguments: level should be the current level minus one, and pointers this_node%reg and this_node%str. The procedure is repeated, always following pointer this_node%reg before this_node%str, until level 1 is reached. At this point, the two pointers are initialized and made to point nowhere (NULLIFIED). By the order of precedence inherent in it, the routine then backs up one level and proceeds to follow pointer this_node%str to the “dead end” at level 1.
In this fashion, the tree is initialized from left ($J$) to right ($J^*$), with the pointer to the subnode $J^*$ of the rightmost node being allocated last. If we refer back to figure 4, the nodes for the tree of this example will be initialized in the following order:

$$\{1,3,7\} \rightarrow \{1,3\} \rightarrow \{1\} \rightarrow \{2\} \rightarrow \{4,6\} \rightarrow \{4\} \rightarrow \{2\}.$$ 

In order for subsequent routines to identify an initialized but unfilled (constituents of node empty) node, BuildNodeTree will set this_node%v (this_node%v%mat(1,1) in BDT2.F90) to zero, upon allocation of pointers.

### A.2 Fill_Tree

A RECURSIVE SUBROUTINE, taking on a pointer of type node as argument. Its function is to traverse the now initialized tree, and using the flag for an unfilled node, fill it by calling Fill_Node.

### A.3 Fill_Node

A RECURSIVE SUBROUTINE, called by Fill_Tree, whose function is to fill the particular node that its pointer argument points to. It is in this routine that the BDT Algorithm proper is applied, modifying the reflection coefficient calculation according to the selected method. Care must be taken when calculating the forward and backward prediction errors, $\varepsilon_K(t)$ and $\eta_K(t)$, before termination of the routine. Each should be calculated over a sufficiently large range of $t$ values ($1 \leq t \leq n + k_m$ for Yule-Walker, and $1 + k_m \leq t \leq n$ for the remaining methods), since subsequent nodes may use them.

### A.4 Print_Node_Tree

With its pointer argument, the RECURSIVE SUBROUTINE Print_Node_Tree will traverse the now completed tree of nodes, and proceed to print the estimated coefficients and white noise variance stored in each node.

### A.5 Causal_Check

This routine is needed in the bivariate program only, in order to ensure the obtained VAR model is causal before proceeding with the likelihood calculations. In the univariate program, this function is performed within
the likelihood calculation routine itself. The strategy is to use the state space representation to write a VAR(p) as a VAR(1), as follows:

Random vectors \{X_t, \ldots, X_{t-k_m}\} from model (1), will satisfy the relationships

\[
\begin{bmatrix}
X_t \\
X_{t-1} \\
X_{t-2} \\
\vdots \\
X_{t-k_m+1}
\end{bmatrix} =
\begin{bmatrix}
0 & 0 & \cdots & 0 & k_1 & \cdots & k_m \\
I_d & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\
0 & I_d & \cdots & \cdots & \cdots & \cdots & 0 \\
\vdots & \vdots & \ddots & \cdots & \cdots & \cdots & \vdots \\
0 & \cdots & I_d & 0
\end{bmatrix}
\begin{bmatrix}
X_{t-1} \\
X_{t-2} \\
X_{t-3} \\
\vdots \\
X_{t-k_m}
\end{bmatrix} +
\begin{bmatrix}
Z_t \\
0 \\
0 \\
\vdots \\
0
\end{bmatrix},
\]

which can be written in the compact form

\[
Y_t = A Y_{t-1} + W_t. \quad (28)
\]

In block matrix form, vectors \(Y_t\) and \(W_t\) have length \(k_m\), while the square matrix \(A\) has dimension \(k_m\). Note that the only nonzero entries of the first block matrix row of \(A\) are \(\{\Phi_K(k_1), \Phi_K(k_2), \ldots, \Phi_K(k_{m-1}), \Phi_K(k_m)\}\), occurring at block matrix column numbers \(\{k_1, k_2, \ldots, k_{m-1}, k_m\}\), respectively. The covariance matrix of \(W_t\) is

\[
\Sigma_W = E \begin{bmatrix}
Z_t \\
0 \\
\vdots \\
0
\end{bmatrix} \begin{bmatrix}
Z_t' \\
0 \\
\vdots \\
0
\end{bmatrix} =
\begin{bmatrix}
\Sigma & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & 0
\end{bmatrix},
\]

where we use \(\Sigma\) in place of \(U_K\). (28) is now a VAR(1) of dimension \(dk_m\), and its causality (and thus that of the original process) can be assessed by determining if all eigenvalues of \(A\) are less than 1 in absolute value.

A.6 Likelihood/Approx_Likelihood

In the univariate program, we compute the exact likelihood in \texttt{SUBROUTINE Likelihood}. The only sizeable difficulty is in evaluating the model autocovariances \(\gamma(0), \ldots, \gamma(k_m)\), accomplished by inverting the Yule-Walker equations. The \(-2\) log likelihood, \(L(\phi_K, \sigma^2)\), for the data \(x_1, \ldots, x_n\), is
then evaluated via the Innovations Algorithm (Brockwell and Davis, 1991, prop. 5.2.2, and equation 8.7.4):

$$L(K; 2) = n \log(2^2) + \sum_{t=1}^{n} \log(r_t^2) + \frac{1}{\sigma^2} \sum_{t=1}^{n} (x_t - \hat{x}_t)^2 / r_{t-1}.$$ 

In the bivariate program, SUBROUTINE Likelihood uses the same approach to compute the likelihood, i.e. the Multivariate Innovations Algorithm (Brockwell and Davis, 1991, prop. 11.4.2, and equation 11.5.5):

$$L(\Phi_K, \Sigma) = nd \log(2\pi) + \sum_{t=1}^{n} \log | V_{t-1} | + \sum_{t=1}^{n} (X_t - \hat{X}_t)'V_{t-1}^{-1}(X_t - \hat{X}_t).$$

Computing the model autocovariance matrices, $\Gamma(1-k_m), \ldots, \Gamma(0), \ldots, \Gamma(k_m-1)$, is a much more formidable task here, but this can be accomplished via the state space formulation of the previous subsection. Transforming the SVAR($K$) to the VAR(1) of equation (28), gives the following solution for the autocovariances $\Gamma_Y(\cdot)$ of the process $\{Y_t\}$:

$$\Gamma_Y(h) = \begin{cases} A \Gamma_Y(h) A' + \Sigma_W, & h = 0 \\ A \Gamma_Y(h - 1), & h > 0 \end{cases},$$

whence we obtain

$$vec(\Gamma_Y(0)) = \left[ I_{d^2k_m^2} - A \otimes A \right]^{-1} vec(\Sigma_W).$$

The required autocovariance matrices can be found in the first block row and column of the $(k_m \times k_m)$ block matrix $\Gamma_Y(0)$, since

$$\Gamma_Y(0) = \begin{bmatrix} \Gamma(0) & \Gamma(1) & \cdots & \Gamma(k_m - 1) \\ \Gamma(-1) & \Gamma(0) & \cdots & \Gamma(k_m - 2) \\ \vdots & \ddots & \ddots & \vdots \\ \Gamma(1 - k_m) & \cdots & \Gamma(-1) & \Gamma(0) \end{bmatrix}.$$

Due to the computational intensity involved in finding $\Gamma_Y(\cdot)$ however, the bivariate routine Likelihood is extremely slow. We opt instead to approximate the autocovariances via the causal representation

$$\Gamma(h) = \sum_{j=0}^{\infty} \Psi_{h+j} \Sigma \Psi'_j,$$
truncating the summation at 100 terms, and computing the likelihood via (2). This “approximate likelihood”, is computed in SUBROUTINE ObjFun. SUBROUTINE ApproxLikelihood not only calls ObjFun in order to compute this approximate likelihood for $\Sigma_{AL}$, but also searches for the white noise covariance matrix that maximizes the likelihood for the given VAR coefficient matrices ($\Sigma_{ML}$). It does so by using $\Sigma_{AL}$ as an initial guess, and by repeated calls to SUBROUTINE Hooke, which employs a direct search algorithm to locate the global minimum of an objective function of several variables (Hooke and Jeeves, 1961).

B Data Sets

B.1 lynx10.tsm

2.42975228000241
2.50650503240487
2.76715586608218
2.94001815500766
3.16879202031418
3.45040308615537
3.59417147911491
3.77400573025821
3.69460519893357
3.411114185509
2.71850168886727
1.99122607569249
2.26481782300954
2.4456042032736
2.61172330800734
3.3588620440587
3.42894429003557
3.53262700122889
3.2610248339924
2.61172330800734
2.17897694729317
1.65321251377534
1.83250891270624
2.32837960343874
2.73719264270474
3.01410032151962
2.58994960132571
1.86332286012046
1.5910646070265
1.69019608002851
1.77085201164214
2.27415784926368
2.57634135020579
3.1126251365907
3.60541279815305
3.5434471800817
2.76863810124761
2.02118929906994
2.1846914308176
2.58771096501891
2.87966920563205
3.11627558758054
3.53970323894783
3.84453930212901
3.80023578932735
3.57909732655264
2.26387267686522
2.53781909507327
2.5820636291171
2.90741136077459
3.14238946611884
3.4334497937616
3.57978359661681
3.4900990050633
3.4749433546539
3.57863920996807
2.82865989653532
1.90848501887865
1.90308998699194
2.03342375548695
2.35983548233989
2.6097289568675
3.05384642685225
3.3859635706007
3.55315454816963
3.46760810558363
B.2 sun2.tsm

\begin{verbatim}
101.000000000000000  28.000000000000000
 66.000000000000000  17.000000000000000
 31.000000000000000  67.000000000000000
 20.000000000000000  92.000000000000000
154.000000000000000  35.000000000000000
 85.000000000000000  24.000000000000000
 38.000000000000000  23.000000000000000
 10.000000000000000  68.000000000000000
 83.000000000000000  132.000000000000000
131.000000000000000  118.000000000000000
 90.000000000000000  67.000000000000000
 60.000000000000000  47.000000000000000
 41.000000000000000  21.000000000000000
 16.000000000000000  6.000000000000000
  4.000000000000000  7.000000000000000
14.000000000000000  34.000000000000000
45.000000000000000  43.000000000000000
48.000000000000000  42.000000000000000
28.000000000000000  10.000000000000000
 8.000000000000000  2.000000000000000
 0.000000000000000E+00  1.000000000000000
  5.000000000000000  12.000000000000000
14.000000000000000  35.000000000000000
46.000000000000000  41.000000000000000
30.000000000000000  24.000000000000000
16.000000000000000  7.000000000000000
 4.000000000000000  2.000000000000000
 8.000000000000000  17.000000000000000
36.000000000000000  50.000000000000000
\end{verbatim}
C  Coded Versions of the BDT Algorithm

C.1  Code for BDT.F90

MODULE tree

! Here we define the data type NODE which will contain

  TYPE node
    INTEGER :: level ! level in tree: top=m, bottom=1
    INTEGER :: lags(26) ! lags for node are stored here
    DOUBLE PRECISION :: phi(26) ! coefficients for node
    DOUBLE PRECISION :: v ! MSE (white noise) for node
    DOUBLE PRECISION :: eps(1:10100) ! the epsilons for the node
    DOUBLE PRECISION :: eta(-99:10000) ! the etas for the node
    TYPE (node), POINTER :: reg, star ! pointers to the regular and
    END TYPE node ! starred subnodes one level down

! These will contain the end results
DOUBLE PRECISION :: topphi(26), topv, acvf(1000)

! Other globals
INTEGER :: n, m, method
DOUBLE PRECISION, ALLOCATABLE :: x(:)
CHARACTER :: stamp*4

CONTAINS

SUBROUTINE make_tree(original_lags)
INTEGER, ALLOCATABLE :: toplags(:)
INTEGER, INTENT (IN) :: original_lags(m)
DOUBLE PRECISION :: x(n)
TYPE (node), POINTER :: top_node

NULLIFY (top_node) !associates top_node so we can use it
ALLOCATE(toplags(m))
toplags=original_lags

! now build the tree of node lags
CALL build_node_tree(m,toplags,top_node)

! now fill the tree, ie. get coeffts and MSE’s of each node
CALL fill_tree(top_node)

! node tree built, so print it
CALL print_node_tree(top_node)

! Likelihood calculation
CALL likelihood(original_lags, topphi(1:m), topv)

DEALLOCATE(toplags)

RETURN
END SUBROUTINE make_tree

!***********************************************************************

RECURSIVE SUBROUTINE build_node_tree(lev,this_lags,this_node)
INTEGER :: i, lev, this_lags(26)
TYPE (node), POINTER :: this_node

! This routine will create a tree of nodes needed to subset Burg model. The level and lags are assigned to each node. Also the MSE of each node is initialized to be zero so that later we'll be able to check which nodes have not yet been filled.

! first time thru' with a fresh node; point to it & make its J and J* pointers point nowhere
IF (.NOT. ASSOCIATED(this_node)) THEN
   ALLOCATE (this_node)
   this_node%level=lev
   this_node%lags(1:lev)=this_lags(1:lev)
! the check for an unfilled node will be that its MSE=0
   this_node%v=0
   NULLIFY (this_node%reg)
   NULLIFY (this_node%star)
END IF

! recursive call to routine with J lags; only if level>1
! lev=this_node%level
IF (this_node%level>1) THEN
   this_lags(1:lev-1)=this_node%lags(1:lev-1)
   CALL build_node_tree(lev-1,this_lags,this_node%reg)
END IF

! recursive call to routine with J* lags; only if level>1
! lev=this_node%level
IF (this_node%level>1) THEN
   this_lags(1:lev-1)=(/(this_node%lags(lev) &
   -this_node%lags(lev-i), i=1,lev-1)/)
   CALL build_node_tree(lev-1,this_lags,this_node%star)
END IF

RETURN
END SUBROUTINE build_node_tree

!**************************************************************************

32
RECURSIVE SUBROUTINE fill_tree(this_node)
    INTEGER :: i
    TYPE (node), POINTER :: this_node

    ! Here we fill in the coeffts and MSE for each node
    IF (this_node%level>1) THEN
        CALL fill_tree(this_node%reg)
        ! if reg node has not been filled, then fill it!
        IF (this_node%reg%v==0) THEN
            CALL fill_node(this_node%reg)
        END IF
        CALL fill_tree(this_node%star)
        ! if star node has not been filled, then fill it!
        IF (this_node%star%v==0) THEN
            CALL fill_node(this_node%star)
        END IF
    END IF

    ! now that whole tree is filled, we can fill top node
    IF (this_node%level==m) THEN
        CALL fill_node(this_node)
    END IF

    RETURN
END SUBROUTINE fill_tree

SUBROUTINE fill_node(this_node)
    INTEGER :: i, km, lev, sum_range(2)
    DOUBLE PRECISION, ALLOCATABLE :: eps_J(:,), eps_Js(:,), eta_J(:,), eta_Js(:,)
    DOUBLE PRECISION :: v_J, v Js, phi_K, phi_Ks, top, sum_en
    DOUBLE PRECISION :: phi_J(26), phi_Js(26), sum_e, sum_n
    TYPE (node), POINTER :: this_node

    ! This is the routine where the real work of building the coeffts, MSEs,!
    ! epsilons and etas is done

! initialize the precursors before applying algo to this node
lev=this_node%level
km=this_node%lags(lev)
ALLOCATE (eps_J(1:n+km), eps_Js(1:n+km), eta_J(1-km:n), eta_Js(1-km:n))
IF (lev==1) THEN
  eps_J = 0.0
  eps_Js = 0.0
  eta_J = 0.0
  eta_Js = 0.0
  eps_J (1:n) = x(1:n)
  eps_Js (1:n) = x(1:n)
  eta_J (1:n) = x(1:n)
  eta_Js (1:n) = x(1:n)
  v_J = SUM(x**2)/n
  v_Js = v_J
ELSE ! we're at a higher level, so use reg and star node info
  phi_J (1:lev-1) = this_node%reg%phi (1:lev-1)
  phi_Js (1:lev-1) = this_node%star%phi (1:lev-1)
  eps_J (1:n+km) = this_node%reg%eps (1:n+km)
  eps_Js (1:n+km) = this_node%star%eps (1:n+km)
  eta_J (1-km:n) = this_node%reg%eta (1-km:n)
  eta_Js (1-km:n) = this_node%star%eta (1-km:n)
  v_J = this_node%reg%v
  v_Js = this_node%star%v
END IF

! Initial conditions set, now apply algo to this node
! First the reflection coefficients
IF (method==1) THEN ! YuWa
  sum_range=(/1,n+km/)
  top=0
  DO i=1,n+km
     top=top+(eps_J(i)*eta_Js(i-km))
  END DO
  phi_K = top/(n*v_Js)
  phi_Ks = top/(n*v_J)
ELSE ! Burg type
  sum_range=(/1+km,n/)
! First get sum of squares and cross squares for eps and eta:
sum_e=0; sum_n=0; sum_en=0
DO i=km+1,n
    sum_en=sum_en + eps_J(i)*eta_Js(i-km)
    sum_e=sum_e + eps_J(i)**2
    sum_n=sum_n + eta_Js(i-km)**2
END DO
!
Now calculate reflection coefficients depending on the method:
SELECT CASE (method)
CASE (2) ! Burg
    phi_K =v_J*(v_J+v_Js)*sum_en/(sum_n*v_J**2 + sum_e*v_Js**2)
    phi_Ks=v_Js*(v_J+v_Js)*sum_en/(sum_n*v_J**2 + sum_e*v_Js**2)
CASE (3) ! Morf
    phi_K =SQR(v_J/(v_Js*sum_e*sum_n))*sum_en
    phi_Ks=SQR(v_Js/(v_J*sum_e*sum_n))*sum_en
CASE (4) ! Nutt
    phi_K =2.0*v_J*sum_en/(v_Js*sum_e+v_J*sum_n)
    phi_Ks=v_Js*phi_K/v_J
END SELECT
END IF
!
Continue with remaining recursions - identical for all algo’s
this_node%phi(lev)=phi_K
IF (lev > 1) THEN
    DO i=1,lev-1
        this_node%phi(i)=phi_J(i)-phi_K*phi_Js(lev-i)
    END DO
END IF
this_node%v=(1-phi_Ks*phi_K)*v_J
!
The eta’s & epsilon’s for posterity:
DO i=sum_range(1),sum_range(2)
    this_node%eps(i) =eps_J(i)-phi_K*eta_Js(i-km)
    this_node%eta(i-km)=eta_J(i-km)-phi_K*eps_Js(i)
END DO

DEALLOCATE (eps_J,eps_Js,eta_J,eta_Js)
RETURN
END SUBROUTINE fill_node
SUBROUTINE print_node_tree(this_node)

INTEGER :: i

TYPE (node), POINTER :: this_node

! Here we print the info in the top node - can also make it print all nodes
! by removing the inmost IF THEN loop

IF (ASSOCIATED(this_node)) THEN
    CALL print_node_tree(this_node%reg)
    IF (this_node%level==m) THEN
        PRINT*, "*******************************************
        PRINT*, "The estimated subset ", stamp, " AR coefficients are:
        DO i=1,this_node%level
            PRINT 10, this_node%lags(i), this_node%phi(i)
        10 FORMAT(" Phi(",I2,"): ",F9.5)
        END DO
        PRINT*, "*******************************************
        PRINT*, stamp, " WN variance estimate : ",this_node%v
    END IF
    CALL print_node_tree(this_node%star)
    IF (this_node%level<m) DEALLOCATE (this_node)
END IF
RETURN
END SUBROUTINE print_node_tree

SUBROUTINE likelihood(lag, phi, s2)

! First computes ACVF of a subset AR model with m coeffts (phi) and lags
! (lag), and sigma^2=s2, into acvf, lags 0 to n-1 acvf(0:n-1). Then it gets
! -2 log Likelihood for the vector of obs x:
! -2log L = n*log(2*pi*s2) + sum(log(r(j))) + resid_ss/s2
! using the innovations algorithm

INTEGER :: i, j, k, km, t
INTEGER :: lag(m), lags(m+1)
DOUBLE PRECISION :: phi(m), phis(m+1), acvf(0:2*lag(m)), Ka(n,n)
DOUBLE PRECISION :: A(lag(m)+1, lag(m)+1), b(lag(m)+1)
DOUBLE PRECISION :: s, s2, pi, resid_ss, xh(n), loglike_wn, loglike_ss
DOUBLE PRECISION :: cond_like, th(1:n-1,0:n-1), r(0:n-1), aicc

km=lag(m)
pi=3.141592654
lags(1) =0
lags(2:m+1)=lag
phis(1) =-1.0
phis(2:m+1)=phi
b=0.0
b(1)=s2
! Solve system A*acvf(0:km) = b, to get acvf(0:km)
DO i=1,km+1
   A(i,:) =0.0
END DO
DO k=0,km
   DO j=0,m
      A(k+1, ABS(k-lags(j+1))+1) = A(k+1, ABS(k-lags(j+1))+1) - phis(j+1)
   END DO
END DO
CALL DLSARG(1+km, A, 1+km, b, 1, acvf(0:km))

! Now get acvf(km+1:2*km) via recursions
DO k=km+1, 2*km
   acvf(k)=0.0
   DO j=1,m
      acvf(k)=acvf(k)+phi(j)*acvf(k-lag(j))
   END DO
END DO

! Form K(.,.) as in (5.3.5)
DO i=1,n
   Ka(i,:) =0.0
END DO
DO i=1,n
   DO j=1,n
      IF (i<=km .AND. j<=km) Ka(i,j)=acvf(abs(i-j))/s2
      IF (min(i,j)<=km .AND. km<max(i,j) .AND. max(i,j)<=2*km) THEN
s=0
DO k=1,m
  s=s+\phi(k)\cdot acvf(abs(lag(k)-abs(i-j)))
END DO
Ka(i,j)=(acvf(abs(i-j))-s)/s^2
END IF
IF (min(i,j)>km .AND. i==j) Ka(i,j)=1.0
END DO
END DO

! Form (5.2.16) recursions
th(:,0)=1.0
r(0)=Ka(1,1)
DO i=1,n-1
  DO k=0,i-1
    s=0
    DO j=0,k-1
      s=s+th(k,k-j)*th(i,i-j)*r(j)
    END DO
    th(i,i-k)=(Ka(i+1,k+1)-s)/r(k)
  END DO
  s=0
  DO j=0,i-1
    s=s+r(j)*th(i,i-j)**2
  END DO
  r(i)=Ka(i+1,i+1)-s
END DO

! Build 1-step predictors (xh’s), and get the resid_ss
xh(1)=0.0
DO k=1,n-1
  IF (k < km) THEN
    xh(k+1)=0.0
    DO j=1,k
      xh(k+1)=xh(k+1)+th(k,j)*(x(k+1-j)-xh(k+1-j))
    END DO
  ELSE ! k >= km
    xh(k+1)=0.0
    DO j=1,m
      xh(k+1)=xh(k+1)+\phi(j)*x(k+1-lag(j))
  END IF
... END DO
... END IF
... END DO
s=0
resid_ss=0
DO j=1,n
  IF (r(j-1)<=0) THEN
    PRINT*, "### NON-CAUSAL MODEL ###"
    RETURN
  END IF
  s=s+log(r(j-1))
  resid_ss=resid_ss+(x(j)-xh(j))**2/r(j-1)
END DO

! calculate cond. likelihood
! -2 log CL = (n-km)*log(s2)+(1/s2)sum_{km+1}^n (x_t-phi_k1*x_{t-k1}-...
! -phi_km*x_{t-km})^2
cond_like=0
DO t=km+1,n
  cond_like=cond_like+DOT_PRODUCT(phi,/(x(t-lag(j)), j=1,m)/))
END DO
cond_like=(n-m)*log(s2)+cond_like/s2

! Finally: -2log Likelihood=loglike: wn means use WN variance estimate:
! SS means use RSS/n variance estimate
loglike_wn = n*log(2.0*pi*s2) + s + resid_ss/s2
loglike_ss = n*log(2.0*pi*resid_ss/n) + s + FLOAT(n)
PRINT *,"RSS/n WN variance estimate: ", resid_ss/n
PRINT","-2 Log Like (",stamp,"): ",loglike_wn
PRINT","-2 Log Like (RSS/n):",loglike_ss
aicc=loglike_ss+2.0*n*(m+1)/FLOAT(n-m-2)
PRINT","AICC (RSS/n): ", aicc
  PRINT*, " s ",s
  PRINT ",","-2 Log Cond Like (YW WN): ", cond_like
  PRINT","~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
RETURN
END SUBROUTINE likelihood

END MODULE tree

39
PROGRAM bdt

! Author: A. Trindade, www.stat.ufl.edu/~trindade/
! This program subset models AR(p)'s using Burg type recursions
! with subset size <=26, km<100, and a max of 10,000 observations.
! For details refer to the paper: "Implementing Modified Burg Algorithms
! in Multivariate Subset Autoregression", by the author.

USE tree

INTEGER :: i, mc
INTEGER, ALLOCATABLE, DIMENSION (:) :: toplags
DOUBLE PRECISION :: y(10000), mean
CHARACTER :: h*24

PRINT*, "%%%%%%%%%%%%%%%% Univariate SAR Modeling Program %%%%%%%%%%%%%%%%%"

! read in the time series
20 write(*,*)
   write(*,22)
22 format(5x,'File name of time series for modelling: ',$)
23 h=' '
   read(*,*) h
   IF (h=='a ') h='lynx10.tsm'
   open(3,file=h,status='old',err=20)
   i=1
25 read(3,*,end=30) y(i)
   i=i+1
   if (i.eq.10002) then
      write(*,6665)
5665 format(3x,'DATA TRUNCATED AFTER FIRST 10000 OBSERVATIONS.')
      i=i-1
      goto 30
   endif
   goto 25
30 n=i-1
   close(3)
now mean-correct the obs
mean=SUM(y)/n
ALLOCATE (x(n))
x=/(y(i),i=1,n)/
PRINT*, "Do you wish to mean-correct the observations (1=yes, 0=no)?"
READ*, mc
IF (mc==1) x=/(y(i)-mean,i=1,n)/
PRINT*, "There are ",n," observations."
PRINT*,"First obs is ",x(1)," last is ",x(n)

Enter how many lags will be modeled
PRINT*, "Enter the number of lags to be modeled (<27):"
READ*, m
ALLOCATE (toplags(m))

read in the lags
PRINT*, "Enter the lags:"
READ*, (toplags(i), i=1,m)
PRINT*,"The lags are: ", (toplags(i), i=1,m)

Enter the method
PRINT*, "Enter the method for obtaining the reflection coefficients:"
PRINT*, "Yule-Walker (1), Burg (2), Vieira-Morf (3), Nuttall-Strand (4):"
READ*, method
SELECT CASE (method)
CASE (1); stamp="YuWa"
CASE (2); stamp="Burg"
CASE (3); stamp="Morf"
CASE (4); stamp="Nutt"
CASE DEFAULT
PRINT*,"Method not in range: ", method
STOP
END SELECT

CALL make_tree(toplags)

END PROGRAM bdt
C.2 Code for BDT2.F90

MODULE tree

TYPE vector
  DOUBLE PRECISION :: vec(2)
END TYPE vector

TYPE matrix
  DOUBLE PRECISION :: mat(2,2)
END TYPE matrix

! Here we define the data type NODE which will contain
TYPE node
  INTEGER :: level ! level in tree: top=m, bottom=1
  INTEGER :: lags(26) ! lags for node are stored here
  TYPE (matrix) :: A(26) ! forward coefficients for node
  TYPE (matrix) :: B(26) ! backward coefficients for node
  TYPE (matrix) :: vf, vb ! forward and backward MSEs
  TYPE (vector) :: eps(1:10100) ! the epsilons for the node
  TYPE (vector) :: eta(-99:10000) ! the etas for the node
  TYPE (node), POINTER :: reg, star ! pointers to the regular and
END TYPE node ! starred subnodes one level down

! These will contain the end results
  TYPE (matrix) :: topA(1:26), topvf

! Other globals
  INTEGER :: n, m, method
  INTEGER, ALLOCATABLE :: orig_lags(:)
  TYPE (vector), ALLOCATABLE :: x(:)
  CHARACTER :: stamp*4

CONTAINS

SUBROUTINE make_tree(toplags, truevf)
  INTEGER :: toplags(m)
  TYPE (node), POINTER :: top_node
  DOUBLE PRECISION :: truevf(3)
  EXTERNAL DEVLRG

NULLIFY (top_node)  !associates top_node so we can use it
ALLOCATE(orig_lags(m))
orig_lags=toplags

! now build the tree of node lags
CALL build_node_tree(m,toplags,top_node)

! now fill the tree, i.e. get coeffts and MSE’s of each node
CALL fill_tree(top_node)

! node tree built, so print it
CALL print_node_tree(top_node)

! undo node tree so that we don’t run out of memory for next runs
! CALL undo_node_tree(top_node)

! Causal check - program will terminate if noncausal solution.
CALL Causal_Check

! Likelihood calculations (exact or approx)
! CALL likelihood("YW WN")
CALL approx_likelihoods(truevf)

RETURN
END SUBROUTINE make_tree

!***********************************************************************

RECURSIVE SUBROUTINE build_node_tree(lev,this_lags,this_node)
   INTEGER :: i, lev, this_lags(26)
   TYPE (node), POINTER :: this_node

! This routine will create a tree of nodes needed to subset Burg model. The
! level and lags are assigned to each node. Also the MSE of each node is
! initialized to be zero so that later we’ll be able to check which nodes
! have not yet been filled.

! first time thru’ with a fresh node; point to it & make its J and J*
! pointers point nowhere
IF (.NOT. ASSOCIATED(this_node)) THEN
  ALLOCATE (this_node)
  this_node%level=lev
  this_node%lags(1:lev)=this_lags(1:lev)
  ! the check for an unfilled node will be the (1,1) entry of vf=0
  this_node%vf%mat(1,1)=0
  NULLIFY (this_node%reg)
  NULLIFY (this_node%star)
END IF

! recursive call to routine with J lags; only if level>1
lev=this_node%level
IF (lev>1) THEN
  this_lags(1:lev-1)=this_node%lags(1:lev-1)
  CALL build_node_tree(lev-1,this_lags,this_node%reg)
END IF

! recursive call to routine with J* lags; only if level>1
lev=this_node%level
IF (lev>1) THEN
  this_lags(1:lev-1)=/(this_node%lags(lev) &
                     -this_node%lags(lev-i), i=1,lev-1)/
  CALL build_node_tree(lev-1,this_lags,this_node%star)
END IF
RETURN
END SUBROUTINE build_node_tree

!***********************************************************************
RECURSIVE SUBROUTINE fill_tree(this_node)
  INTEGER :: i
  TYPE (node), POINTER :: this_node

  ! Here we fill in the coeffts and MSE for each node

  IF (this_node%level>1) THEN
    CALL fill_tree(this_node%reg)
  ! if reg node has not been filled, then fill it!
    IF (this_node%reg%vf%mat(1,1)==0) THEN

44
CALL fill_node(this_node%reg)
END IF

CALL fill_tree(this_node%star)

! if star node has not been filled, then fill it!
IF (this_node%star%vf%mat(1,1)==0) THEN
  CALL fill_node(this_node%star)
END IF
END IF

! now that whole tree is filled, we can fill top node
IF (this_node%level==m) THEN
  CALL fill_node(this_node)
END IF

RETURN
END SUBROUTINE fill_tree

***********************************************************************
SUBROUTINE fill_node(this_node)
  INTEGER :: i, j, t, km, lev, range(2)
  TYPE (vector), ALLOCATABLE :: eps_J(:), eps_Js(:), eta_J(:), eta_Js(:)
  DOUBLE PRECISION, DIMENSION(2,2) :: vf_J, vf_Js, A_K, B_Ks, sen, see, Id
  DOUBLE PRECISION, DIMENSION(2,2) :: vfinv, vfsinv, B, A_Ks, B_K, vb_J, vb_Js
  DOUBLE PRECISION, DIMENSION(2,2) :: vbinv, vbsinv, snn, R, see2, snn2
  DOUBLE PRECISION, DIMENSION(2,2) :: U2, V2, AA, CC, tV
  DOUBLE PRECISION, DIMENSION(4,4) :: term1, term1_left, term1_right, A
  TYPE (matrix) :: A_J(26), B_Js(26), A_Js(26), B_J(26)
  TYPE (node), POINTER :: this_node

  ! This is the routine where the real work of building the coeffts, MSEs,
  ! epsilons and etas is done

  ! initialize the precursors before applying algo to this node
  lev=this_node%level
  km=this_node%lags(lev)
  ALLOCATE(eps_J(1:n+km), eps_Js(1:n+km), eta_J(1-km:n), eta_Js(1-km:n))
  IF (lev==1) THEN
DO t=n+1,n+km
    eps_J(t)%vec =0; eps_Js(t)%vec=0
END DO
DO t=1-km,0
    eta_J(t)%vec =0; eta_Js(t)%vec=0
END DO
eps_J (1:n)=x(1:n)
eps_Js(1:n)=x(1:n)
eta_J (1:n)=x(1:n)
eta_Js(1:n)=x(1:n)
DO i=1,2
    DO j=1,2
        vf_J(i,j)=DOT_PRODUCT(x%vec(i),x%vec(j))/n
    END DO
END DO
vf_Js =vf_J
vb_Js =vf_J
vb_J =vf_J
ELSE !we're at a higher level, so use reg and star node info
    A_J (1:lev-1) =this_node%reg%A(1:lev-1)
    A_Js(1:lev-1) =this_node%star%A(1:lev-1)
    B_J(1:lev-1) =this_node%reg%B(1:lev-1)
    B_Js(1:lev-1) =this_node%star%B(1:lev-1)
    eps_J (1:n+km) =this_node%reg%eps(1:n+km)
    eps_Js(1:n+km) =this_node%star%eps(1:n+km)
    eta_J (1-km:n) =this_node%reg%eta(1-km:n)
    eta_Js(1-km:n) =this_node%star%eta(1-km:n)
    vf_J =this_node%reg%vf%mat
    vf_Js =this_node%star%vf%mat
    vb_Js =this_node%star%vb%mat
    vb_J =this_node%reg%vb%mat
END IF

! Initial conditions set, now apply algo to this node
! First build A_K(km) ------------------------------------------
range=(/1,n+km/)
IF (method>1) range=(/1+km,n/)
DO i=1,2
    see(i,:)=(/0.0D+00,0.0D+00/)
    sen(i,:)=(/0.0D+00,0.0D+00/)

46
snn(i,:)=(/0.0D+00,0.0D+00/)
END DO
DO t=range(1),range(2)
  DO i=1,2
    DO j=1,2
      see(i,j)=see(i,j)+eps_J(t)%vec(i)*eps_J(t)%vec(j)
      sen(i,j)=sen(i,j)+eps_J(t)%vec(i)*eta_Js(t-km)%vec(j)
      snn(i,j)=snn(i,j)+eta_Js(t-km)%vec(i)*eta_Js(t-km)%vec(j)
    END DO
  END DO
END DO
END DO
! form I
Id(1,:)=(/1.0D+00,0.0D+00/)
Id(2,:)=(/0.0D+00,1.0D+00/)
! get the inverse of vf_J, put into vfinv
CALL DLINRG(2,vf_J,2,vfinv,2)
SELECT CASE (method)
  CASE (1)
    ! get the inverse of vb_Js, put into vbsinv
    CALL DLINRG(2,vb_Js,2,vbsinv,2)
    ! finally, get A_K(km) & B_Ks(km)
    A_K=MATMUL(sen, vbsinv)/FLOAT(n)
    B_Ks=MATMUL(TRANSPOSE(sen), vfinv)/FLOAT(n)
  CASE (2)
    B=sen+MATMUL(MATMUL(vfinv,sen),vb_Js)
    CALL KRON(2,snn,Id,term1_left)
    CALL KRON(2,MATMUL(vb_Js,vb_Js),MATMUL(MATMUL(vfinv,see),vfinv), &
    & term1_right)
    term1=term1_left+term1_right
    CALL DLINRG(4,term1,4,A,4)
    ! finally, build elements of A_K piecemeal
    DO i=1,2
      A_K(i,1)=A(i,1)*B(1,1)+A(i,2)*B(2,1)+A(i,3)*B(1,2)+A(i,4)*B(2,2)
    END DO
    DO i=3,4
      A_K(i-2,2)=A(i,1)*B(1,1)+A(i,2)*B(2,1)+A(i,3)*B(1,2)+A(i,4)*B(2,2)
    END DO
    ! Finished building A_K ----------- now get B_Ks from it
    B_Ks=MATMUL(MATMUL(vb_Js,TRANSPOSE(A_K)),vfinv)
  CASE(3)
CALL Matrix_Power(vf_J, 5.0D-1, U2)
CALL Matrix_Power(vb Js, -5.0D-1, V2)
CALL Matrix_Power(see, -5.0D-1, see2)
CALL Matrix_Power(snn, -5.0D-1, snn2)
R=MATMUL(MATMUL(see2, sen), snn2)
A_K=MATMUL(MATMUL(U2, R), V2)
B_Ks=MATMUL(MATMUL(vb Js, TRANSPOSE(A_K)), vfinv)

CASE(4)
CALL DLINRG(2, vb Js, 2, V2, 2)  ! V2=(V_J*)^-1
U2=vfinv  ! U2=(U_J)^-1
B=MATMUL(snn, V2)
AA=MATMUL(see, U2)
CALL KRON(2, Id, AA, term1_left)
CALL KRON(2, B, Id, term1_right)
term1=term1_left+term1_right
CALL DLINRG(4, term1, 4, A, 4)
CC=2.0*sen
! Now vec(R) = A . vec(CC)
B=CC
! finally, build elements of R piecemeal
DO i=1,2
   R(i,1)=A(i,1)*B(1,1)+A(i,2)*B(2,1)+A(i,3)*B(1,2)+A(i,4)*B(2,2)
END DO
DO i=3,4
   R(i-2,2)=A(i,1)*B(1,1)+A(i,2)*B(2,1)+A(i,3)*B(1,2)+A(i,4)*B(2,2)
END DO
! Finished building R -------------- now get A_K & B_Ks from it
A_K=MATMUL(R, V2)
B_Ks=MATMUL(MATMUL(vb Js, TRANSPOSE(A_K)), vfinv)
END SELECT
! now get A_Ks, just swap star and nostar
DO i=1,2
   see(i,:)=(/0.0D+00,0.0D+00/)
   sen(i,:)=(/0.0D+00,0.0D+00/)
   snn(i,:)=(/0.0D+00,0.0D+00/)
END DO
DO t=range(1), range(2)
   DO i=1,2
      DO j=1,2
         48
see(i,j)=see(i,j)+\varepsilon_Js(t)vec(i)\vec(i)*\varepsilon_Js(t)vec(j)

\text{sen}(i,j)=\text{sen}(i,j)+\varepsilon_Js(t)vec(i)*\eta_J(t-km)vec(j)

\text{snn}(i,j)=\text{snn}(i,j)+\eta_J(t-km)vec(i)\vec(i)*\eta_J(t-km)vec(j)

\text{END DO}

\text{END DO}

\text{END DO}

\text{! get the inverse of \text{vf}_J, put into \text{vfsinv}}
\text{CALL DLINRG(2,\text{vf}_J,2,\text{vfsinv},2)}

\text{SELECT CASE (method)}

\text{CASE (1)}
\text{! get the inverse of \text{vb}_J, put into \text{vbinv}}
\text{CALL DLINRG(2,\text{vb}_J,2,\text{vbinv},2)}
\text{! finally, get A}_Ks(km) & B_K(km)
\text{A}_Ks=MATMUL(\text{sen}, \text{vbinv})/FLOAT(n)
\text{B}_K=MATMUL(TRANSPOSE(\text{sen}), \text{vfsinv})/FLOAT(n)

\text{CASE (2)}
\text{B=sen+MATMUL(MATMUL(vfsinv,\text{sen}),\text{vb}_J)}
\text{CALL KRON(2,\text{snn},Id,term1_left)}
\text{CALL KRON(2,MATMUL(vb_J,vb_J),MATMUL(MATMUL(vfsinv,see),vfsinv), &}
\text{ & \text{term1_right})}
\text{\text{term1=}term1_left+term1_right}
\text{CALL DLINRG(4,term1,4,A,4)}
\text{! finally, build elements of A}_Ks \text{ piecemeal}
\text{DO i=1,2}
\text{A}_Ks(i,1)=A(i,1)*B(1,1)+A(i,2)*B(2,1)+A(i,3)*B(1,2)+A(i,4)*B(2,2)
\text{END DO}
\text{DO i=3,4}
\text{A}_Ks(i-2,2)=A(i,1)*B(1,1)+A(i,2)*B(2,1)+A(i,3)*B(1,2)+A(i,4)*B(2,2)
\text{END DO}
\text{! Finished building A}_Ks \text{ ---------------}
\text{! now get B}_K \text{ from it}
\text{B}_K=MATMUL(MATMUL(\text{vb}_J,TRANSPOSE(A)_Ks),\text{vfsinv})

\text{CASE(3)}
\text{CALL Matrix\_Power(\text{vf}_J, 5.0D-1, U2)}
\text{CALL Matrix\_Power(\text{vb}_J, -5.0D-1, V2)}
\text{CALL Matrix\_Power(\text{see}, -5.0D-1, see2)}
\text{CALL Matrix\_Power(\text{snn}, -5.0D-1, snn2)}
\text{R=MATMUL(MATMUL(see2,\text{sen}),snn2)}
\text{A}_Ks=MATMUL(MATMUL(U2,R),V2)
\text{B}_K=MATMUL(MATMUL(\text{vb}_J,TRANSPOSE(A)_Ks),\text{vfsinv})
CASE(4)
    CALL DLINRG(2,vb_J,2,V2,2) ! V2=(V_J)^-1
    U2=vfsinv ! U2=(U_J*)^-1
    B=MATMUL(snn,V2)
    AA=MATMUL(see,U2)
    CALL KRON(2,Id,AA,term1_left)
    CALL KRON(2,B,Id,term1_right)
    term1=term1_left+term1_right
    CALL DLINRG(4,term1,4,A,4)
    CC=2.0*sen
    ! Now vec(R) = A . vec(CC)
    B=CC
    ! finally, build elements of R piecemeal
    DO i=1,2
        R(i,1)=A(i,1)*B(1,1)+A(i,2)*B(2,1)+A(i,3)*B(1,2)+A(i,4)*B(2,2)
    END DO
    DO i=3,4
        R(i-2,2)=A(i,1)*B(1,1)+A(i,2)*B(2,1)+A(i,3)*B(1,2)+A(i,4)*B(2,2)
    END DO
    ! Finished building R -------------- now get A_Ks & B_K from it
    A_Ks=MATMUL(R,V2)
    B_K=MATMUL(MATMUL(vb_J,TRANSPOSE(A_Ks)),vfsinv)
END SELECT

! assign A_K and B_K to correct node place
    this_node%A(lev)%mat=A_K
    this_node%B(lev)%mat=B_K
! Now do eqtns 1.16 and 1.18
    IF (lev > 1) THEN
        DO i=1,lev-1
            this_node%A(i)%mat=A_J(i)%mat-MATMUL(A_K,B_Js(lev-i)%mat)
            this_node%B(i)%mat=B_J(i)%mat-MATMUL(B_K,A_Js(lev-i)%mat)
        END DO
    END IF
END

! set MSE’s, eqtns 1.19 and 1.20
    this_node%vf%mat=MATMUL((Id-MATMUL(A_K,B_Ks)),vf_J)
    this_node%vb%mat=MATMUL((Id-MATMUL(B_K,A_Ks)),vb_J)
! Set epsilons and etas
DO i=range(1),range(2)
   this_node%eps(i)%vec = eps_J(i)%vec - MATMUL(A_K, eta_Js(i-km)%vec)
   this_node%eta(i-km)%vec = eta_J(i-km)%vec - MATMUL(B_K, eps_Js(i)%vec)
END DO

DEALLOCATE(eps_J, eps Js, eta_J, eta Js)

RETURN
END SUBROUTINE fill_node

!**********************************************************************
RECURSIVE SUBROUTINE print_node_tree(this_node)
 INTEGER :: i, km, t, j
 TYPE (node), POINTER :: this_node

! Here we print the info in the top node - can also make it print all nodes
! by removing the inmost IF THEN loop
IF (ASSOCIATED(this_node)) THEN
   CALL print_node_tree(this_node%reg)
   IF (this_node%level==m) THEN
      PRINT*, "***************************************************"
      PRINT*, "Estimated subset ", stamp, " coefficient matrices:" 
      DO i=1,this_node%level
         PRINT*, "Phi(", this_node%lags(i), "):"
         CALL DWRRRL('',2,2,this_node%A(i)%mat,2,0,'(F20.6)','','NONE','NONE')
      END DO
      PRINT*, "***************************************************"
      PRINT*, "Estimated ", stamp, " (forward) WN covariance matrix:"
      CALL DWRRRL('',2,2,this_node%vf%mat,2,0,'(F20.6)','','NONE','NONE')
      PRINT*, " 
   END IF
   PRINT*, " 
PRINT*, "***************************************************"
PRINT*, "Estimated ", stamp, " (forward) WN covariance matrix:"
CALL DWRRRL('',2,2,this_node%vf%mat,2,0,'(F20.6)','','NONE','NONE')
PRINT*, " 
! Store results for likelihood calcs
   topA(1:m)=this_node%A(1:m)
   topvf=this_node%vf
END IF

51
CALL print_node_tree(this_node%star)
IF (this_node%level<m) DEALLOCATE (this_node)
END IF

RETURN
END SUBROUTINE print_node_tree

!***********************************************************************
SUBROUTINE Matrix_Power(A, pow, B)
! Raises (2 by 2) matrix A to a real power pow, result into matrix B.

INTEGER :: ipath, irank
DOUBLE PRECISION :: pow, tol, A(2,2), B(2,2), D(2,2), U(2,2), V(2,2), eig(2)
ipath=11
tol=1.0D-10

CALL DLSVRR(2, 2, A, 2, ipath, tol, irank, eig, U, 2, V, 2)
D=0.0
D(1,1)=eig(1)**ABS(pow); D(2,2)=eig(2)**ABS(pow)
B=MATMUL(MATMUL(U,D),TRANSPOSE(V))
IF (pow<0) CALL DLINRG(2,B,2,B,2)
RETURN
END SUBROUTINE Matrix_Power

!***********************************************************************
SUBROUTINE KRON (d,A,B,C)
! computes the Kronecker product of square matrices A and B (dim=d), 
! puts into C (dim=d^2)

INTEGER :: d, i, j, k, l, row, col
DOUBLE PRECISION :: A(d,d), B(d,d), C(d**2,d**2)

DO i=1,d
    DO k=1,d
        row=d*(i-1)+k
        DO j=1,d
            C(row,j)=A(i,k)**B(j)
        END DO
    END DO
END DO

52
DO l=1,d
    col=d*(j-1)+l
    C(row,col)=A(i,j)*B(k,l)
END DO
END DO
END DO
END DO
RETURN
END SUBROUTINE KRON

SUBROUTINE Causal_Check
! Checks for causality of obtained solution. If noncausal, program will !
! terminate without likelihood computations.

INTEGER :: i, j, k, t, km
INTEGER :: lag(m)
TYPE (matrix) :: phi(m)
DOUBLE PRECISION, ALLOCATABLE :: A(:, :)
COMPLEX (KIND=8), ALLOCATABLE :: eig(:)
! kind=8 above specifies that eig's entries be double precision (LIB only).
LOGICAL :: causal

lag=orig_lags
phi=topA(1:m)
km=lag(m)
ALLOCATE (A(2*km, 2*km), eig(2*km))

! Build the matrix A ie. can write a VAR(p) as a VAR(1).
A=0
DO  i=1,2*km
    IF (i<3) THEN
        DO j=1,m
            A(i,2*lag(j)-1)=phi(j)%mat(i,1)
            A(i,2*lag(j)) =phi(j)%mat(i,2)
        END DO
    ELSE ! i=>3
        A(i,i-2)=1.0
END IF
END DO

! For the Phi's to be causal, simply check that all 2*km eigenvalues
! of A are < 1 in absolute value.
CALL DEVLRG(2*km, A, 2*km, eig)
causal=.TRUE.
DO i=1,2*km
   IF (ABS(eig(i)) >= 1.0) THEN !solution non-causal
      causal=.FALSE.
      EXIT
   END IF
END DO

IF (causal) THEN !if solution causal, return control to main prog.
DEALLOCATE (A, eig)
RETURN
ELSE
   !solution non-causal, prog. will terminate.
   PRINT*,"%%% NON-CAUSAL SOLUTION. PROGRAM WILL TERMINATE. %%%"
   PRINT*,"******************************************************************************
   DEALLOCATE (A, eig)
   STOP
END IF

END SUBROUTINE Causal_Check

******************************************************************************

SUBROUTINE likelihood(what)
! Exact likelihood calculation: very, very slow...
! First computes ACVF of a 2d subset AR model with m coeffts (phi) and lags
! (lag), sigma^2=s2, into G(1-2km),...,G(-1),G(0),G(1),...,G(km-1).
! Then it gets -2 log Likelihood for the vector of obs x: eqtn (11.5.5)
! in Yellow book, using the multivariate innovations algorithm.

INTEGER :: i, j, k, t, km, drow
INTEGER :: lag(m)
TYPE (vector) :: xh(n)
TYPE (matrix) :: phi(m), s2, V(0:n-1), Vi(0:n-1)
TYPE (matrix) :: Ka(n,n), Th(n-1,n-1)
DOUBLE PRECISION :: Z2(2,2), Tp(2,2), SumLogDetV, Term3, Like, temp(2)
DOUBLE PRECISION :: tempv(2), DetV
TYPE (matrix), ALLOCATABLE :: G(:)
DOUBLE PRECISION, ALLOCATABLE :: A(:,,:), B(:,,:), C(:,,:), D(:,:)
CHARACTER :: what*7

lag=orig_lags
phi=topA(1:m)
s2=topvf
km=lag(m)
pi=3.141592654
Z2=0.0
ALLOCATE (G(1-2*km:km-1),A(2*km,2*km),B(4*km**2,4*km**2),
         C(4*km**2,4*km**2),D(4*km**2,4*km**2))

! Begin --- building the covariances Gamma(h)
! Build the matrix A
A=0
DO i=1,2*km
   IF (i<3) THEN
      DO j=1,m
         A(i,2*lag(j)-1)=phi(j)%mat(i,1)
         A(i,2*lag(j)) =phi(j)%mat(i,2)
      END DO
   ELSE ! i=>3
      A(i,i-2)=1.0
   END IF
END DO
! B=A kronecker A
CALL KRON(2*km,A,A,B)
! Form C = I - B
C=-B
DO i=1,4*km**2
   C(i,i)=1.0-B(i,i)
END DO
! D = inv(C)
CALL DLINRG(4*km**2, C, 4*km**2, D, 4*km**2)
! vec G_y(0) = D vec(s2). This forms G(1-km),...,G(-1),G(0),G(1),...,G(km-1)
DO k=0,km-1
   DO i=1,2
      55
DO j=1,2
    drow=2*k+i+2*(j-1)*km
    G(-k)%mat(i,j)=D(drow,1)*s2%mat(1,1)+D(drow,2*km+2)*s2%mat(2,2) &
                    +(D(drow,2)+D(drow,2*km+1))*s2%mat(1,2)
END DO
END DO
G(k)%mat=TRANSPOSE(G(-k)%mat)
END DO
!
! Now compute G(-km),...,G(-(2km-1))
DO k=km,2*km-1
    G(-k)%mat=0
    DO j=1,m
        G(-k)%mat=G(-k)%mat+MATMUL(G(lag(j)-k)%mat,TRANSPOSE(phi(j)%mat))
    END DO
END DO
!
! print
! DO k=1-2*km,2*km-1
!   PRINT*,"Gamma(",k,")":"
!   CALL DWRRRL('',2,2, G(k)%mat,2, 0,'(W20.6)','NONE','NONE')
! END DO
!
! Begin --- building the K(i,j)'s, recursions (11.4.27)
DO i=1,n
    DO j=1,n
        Ka(i,j)%mat=Z2
    END DO
END DO
DO i=1,n
    DO j=i,n
        IF (j<=km) Ka(i,j)%mat=G(i-j)%mat
        IF (i<=km .AND. km<j .AND. j<=2*km) THEN
            Ka(i,j)%mat=G(i-j)%mat
            DO k=1,m
                Ka(i,j)%mat=Ka(i,j)%mat-MATMUL(G(i-j+lag(k))%mat, &
                                             TRANSPOSE(phi(k)%mat))
            END DO
        END IF
        IF (i==j .AND. i>km) Ka(i,j)%mat=s2%mat
    END DO
END DO
END IF
IF (i==j .AND. i>km) Ka(i,j)%mat=s2%mat
END DO
END DO
DO j=1,n
  DO i=j+1,n
    Ka(i,j)%mat=TRANSPOSE(Ka(j,i)%mat)
  END DO
END DO

! Begin --- building the theta(i,j)'s & V's, recursions (11.4.23)
V(0)%mat=Ka(1,1)%mat

! Store inverses of V, the Vi's
CALL DLINRG(2, V(0)%mat, 2, Vi(0)%mat, 2)

! Initialize & Keep total of sum of log(det(V))'s
DetV=V(0)%mat(1,1)*V(0)%mat(2,2)-V(0)%mat(2,1)*V(0)%mat(1,2)
SumLogDetV=log(DetV)
DO t=1,n-1
  DO k=0,t-1
    Tp=0
    DO j=0,k-1
      Tp=Tp+MATMUL(MATMUL(Th(t,t-j)%mat,V(j)%mat),TRANSPOSE(Th(k,k-j)%mat))
    END DO
    Th(t,t-k)%mat=MATMUL((Ka(t+1,k+1)%mat-Tp),Vi(k)%mat)
  END DO
  Tp=0
  DO j=0,t-1
    Tp=Tp+MATMUL(MATMUL(Th(t,t-j)%mat,V(j)%mat),TRANSPOSE(Th(t,t-j)%mat))
  END DO
  Get V's, their inverses Vi's, and keep total of sum of log(det(V))'s
  V(t)%mat=Ka(t+1,t+1)%mat-Tp
  CALL DLINRG(2, V(t)%mat, 2, Vi(t)%mat, 2)
  DetV=V(t)%mat(1,1)*V(t)%mat(2,2)-V(t)%mat(2,1)*V(t)%mat(1,2)
  SumLogDetV=SumLogDetV+log(DetV)
END DO

! Get the one step predictors xh's, (11.4.28)
  xh(1)%vec=0
  Term3=DOT_PRODUCT(x(1)%vec,MATMUL(Vi(0)%mat,x(1)%vec))
DO t=1,n-1
  xh(t+1)%vec=0
  IF (t<km) THEN
    DO j=1,t
      xh(t+1)%vec=xh(t+1)%vec+MATMUL(Th(t,j)%mat,x(t+1-j)%vec-xh(t+1-j)%vec)
    END DO
  END IF
ELSE ! t=>km
   DO j=1,m
       xh(t+1)%vec=xh(t+1)%vec+MATMUL(phi(j)%mat,x(t+1-lag(j))%vec)
   END DO
   END IF

! Keep total of Term3 = \sum_{t=1}^{n} [(x(t)-xh(t))'Vi(t-1)(x(t)-xh(t))] \n   Term3=Term3+DOT_PRODUCT(x(t+1)%vec-xh(t+1)%vec, & \n      MATMUL(Vi(t)%mat,x(t+1)%vec-xh(t+1)%vec))

END DO

! Finally: -2 log Likelihood = Like
Like=2.0*n*log(2.0*pi)+SumLogDetV+Term3

DO t=0,km-1
   PRINT*,"Gamma(",t,":")
   CALL DWRRL('',2,2,G(t)%mat,2,0,'(F20.6)','NONE','NONE')
END DO

PRINT*,-2 Log Like (",what," : ", Like
! PRINT*,"AICC (Burg WN): ", Like+4.0*n*(4.0*km+1)/(2.0*(n-1)-4.0*km)

DEALLOCATE (G,A,B,C,D)

RETURN
END SUBROUTINE likelihood

!***********************************************************************
SUBROUTINE approx_likelihoods(truevf)
! Computes approx -2 log likelihood for the algorithm obtained
! Phi's (topA), and WN variance (topvf).
! Also computes WN variance estimate (RSS/n) starting with topvf WN estimate,
! that minimizes the -2 log likelihood for the algorithm obtained Phi's.

INTEGER :: i, maxitn
DOUBLE PRECISION :: ff, tzz(3), step, oldff, truevf(3)
DOUBLE PRECISION :: like1, like2, like

! Get likelihood for the WN variance estimate
tzz(1)=topvf%mat(1,1); tzz(2)=topvf%mat(2,2); tzz(3)=topvf%mat(1,2)
CALL lad_fun(3, tzz, oldff)

! PRINT*,"AICC (Burg WN): ", ff+4.0*n*(4.0*km+1)/(2.0*(n-1)-4.0*km)
PRINT*,-2 Log Like (".,stamp,"): ", oldff

! Now Get WN estimate that maximizes likelihood, for the algorithm Phi's.
! This is equivalent to RSS/n in one dimension. First start search with
! topvf estimate from algo
step=0.1
CALL Hooke(3, tzz, step, ff)
DO WHILE (ABS(ff-oldff) > 0.0001)
   oldff=ff
   step=step/10.0
   CALL Hooke(3, tzz, step, ff)
END DO
like1=MIN(ff, oldff)

! then use truevf (if it exists)
IF (truevf(1)>0) THEN
   CALL lad_fun(3, truevf, oldff)
   step=0.1
   CALL Hooke(3, truevf, step, ff)
   DO WHILE (ABS(ff-oldff) > 0.0001)
      oldff=ff
      step=step/10.0
      CALL Hooke(3, truevf, step, ff)
   END DO
   like2=MIN(ff, oldff)
ELSE
   like2=1.0D30
END IF

! Now take lowest like as the RSS/n
IF (like1 > like2) THEN
   like=like2
tzz=truevf
ELSE ! like1 smaller
   like=like1
END IF
PRINT*,-2 Log Like (RSS/n): ", like
SUBROUTINE approx_likelihoods
!
get the RSS/n matrix
topvf%mat(1,1)=tzz(1); topvf%mat(2,2)=tzz(2)
topvf%mat(1,2)=tzz(3); topvf%mat(2,1)=tzz(3)
PRINT*," 
PRINT*,"Estimated RSS/n (forward) WN covariance matrix:"
CALL DWRRL('',2,2,topvf%mat,2,0,'(F20.6)','NONE','NONE')
PRINT*,"%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"
RETURN
END SUBROUTINE approx_likelihoods

SUBROUTINE lad_fun(nn, zz, ff)
! *** Objective function to go with H&J routine ***
! Computes approx -2 log likelihood for the given Phi's (phi), and the 3
! components of the WN variance (zz). Uses recursions 11.3.12 (YB) to find
! Psi's. G(h) found by truncated summation h=0,...,l (instead of infty).

INTEGER :: i, j, k, t, km, l, h, nn
INTEGER :: lag(m)
TYPE (matrix) :: phi(m), s2, s2i
DOUBLE PRECISION :: t1, t2, t3, t4, tvec(2), zz(*), ff
DOUBLE PRECISION, ALLOCATABLE :: GK(:,,:), GKi(:,,:), y(:)
TYPE (matrix), ALLOCATABLE :: G(:,), Psi(:,), Phy(:,)

l=100 ! the truncation for the ACVF's from the PSI's
lag=orig_lags
phi=topA(1:m)
km=lag(m)
pi=3.141592654
!
Build in constraints for +ve def. WN:
IF ((zz(1)<0).OR.(zz(2)<0).OR.(zz(1)*zz(2)<zz(3)**2)) THEN
  ff=1.0D30
  RETURN
END IF
!
IF (zz(1)<0.0) zz(1)=ABS(zz(1))
IF (zz(2)<0.0) zz(2)=ABS(zz(2))
IF (zz(1)*zz(2)<zz(3)**2) zz(3)=SQRT(zz(1)*zz(2))/2.0

Build the WN, s2
s2%mat(1,1)=zz(1); s2%mat(2,2)=zz(2)
s2%mat(2,1)=zz(3); s2%mat(1,2)=zz(3)

ALLOCATE (G(0:km-1), GK(2*km,2*km), Psi(0:km-1+l), Phy(1:km))
ALLOCATE (y(2*km), GKi(2*km,2*km))

! Need to form Phy=0 except at lag(1:m)
DO h=1,km
   Phy(h)%mat(1,:)=(/0.0,0.0/); Phy(h)%mat(2,:)=(/0.0,0.0/)
END DO
DO t=1,m
   Phy(lag(t))%mat=phi(t)%mat
END DO

! Now get the Psi’s
Psi(0)%mat(1,:)=(/1.0,0.0/); Psi(0)%mat(2,:)=(/0.0,1.0/)
DO j=1,km-1+l
   Psi(j)%mat(1,:)=(/0.0,0.0/); Psi(j)%mat(2,:)=(/0.0,0.0/)
   DO t=1,MIN(j,km)
      Psi(j)%mat=Psi(j)%mat+MATMUL(Phy(t)%mat,Psi(j-t)%mat)
   END DO
END DO

! Now compute covariance matrices
DO h=0,km-1
   G(h)%mat(1,:)=(/0.0,0.0/); G(h)%mat(2,:)=(/0.0,0.0/)
   DO j=0,1
      G(h)%mat=G(h)%mat+MATMUL(MATMUL(Psi(h+j)%mat,s2%mat), &
      TRANSPOSE(Psi(j)%mat))
   END DO
   PRINT*,"Truncated Gamma("h","), with l="l",":
   CALL DWRRRL(’’,2,2,G(h)%mat,2,0,’(F20.6)’,’NONE’,’NONE’)  
END DO

! Form Big covariance matrix GK (symmetric):
DO i=1,km
DO j=1,i
  DO h=0,1
    DO k=0,1
      GK(2*i-h,2*j-k)=G(i-j)\%mat(2-h,2-k)
      GK(2*j-k,2*i-h)=GK(2*i-h,2*j-k)
    END DO
  END DO
END DO
END DO
END DO

! Form term4
  t4=0.0
  CALL DLINDS(2, s2\%mat, 2, s2i\%mat, 2)
  DO t=1+km,n
    tvec=0.0
    DO k=1,m
      tvec=tvec+MATMUL(phi(k)\%mat,x(t-lag(k))\%vec)
    END DO
    tvec=x(t)\%vec-tvec
    t4=t4+DOT_PRODUCT(tvec,MATMUL(s2i\%mat,tvec))
  END DO
END DO

! Form t3
  CALL DLINDS(2*km, GK, 2*km, GKi, 2*km)
  DO j=1,km
    DO k=0,1
      y(2*j-k)=x(j)\%vec(2-k)
    END DO
  END DO
  t3=DOT_PRODUCT(y,MATMUL(GKi,y))

! Form t2
  t2=(n-km)*LOG(s2%mat(1,1)*s2%mat(2,2)-s2%mat(1,2)**2)

! Form t1
  CALL DLFTDS(2*km,GK,2*km,GKi,2*km)
  t1=0.0
  DO h=1,2*km
t1=t1+LOG(GKi(h,h))
END DO

! Now put it all together
ff=2.0*n*log(2.0*pi)+t1+t2+t3+t4

DEALLOCATE (G, GK, GKi, Psi, Phy, y)

RETURN
END SUBROUTINE lad_fun

!*********************************************************************
! subroutine optimization using Hooke and Jeeves
SUBROUTINE Hooke(p,AR,er,fv)

    INTEGER p
    DOUBLE PRECISION AR(p),er,fv

    INTEGER flg,fi,ik,m
    DOUBLE PRECISION x1(p),x2(p),c(p),bx(p)
    DOUBLE PRECISION ac, min,bmin
    intent(inout) :: AR,er

    m=p
    ac=.4*er
    fi=0
    ik=0
    ij=0
    iq=1
    x1=AR

    do 3075 i=1,m
    3075 c(i)=x1(i)
    ! 3080 if(iq.eq.0)then
    !       write(*,*) ' <Computing Gaussian likelihood>'
    !   endif
    call lad_fun(p,c,fv)
    min=fv
    3085 if(iq.eq.0)goto 3640

63
goto 3600
!
! Exploratory moves
!
3100 do 3105 j=1,m
3105 x2(j)=x1(j)
do 3160 i=1,m
d0 3115 j=1,m
3115 c(j)=x2(j)
c(i)=x1(i)+er
call lad_fun(p,c,fv)
3120 if(fv.lt.min) then
   min=fv
   x2(i)=c(i)
endif
c(i)=x1(i)-er
call lad_fun(p,c,fv)
3130 if(fv.lt.min) then
   min=fv
   x2(i)=c(i)
endif
3160 continue
flg=1
do 3180 i=1,m
if(x1(i).ne.x2(i))flg=0
3180 continue
if(flg.eq.0)goto 3195
   er=er/2
if(er.lt.ac)fi=1
if(ik.eq.0)goto 3285
if(ik.eq.1)goto 3605
goto 3100
3195 if(ik.eq.0)goto 3285
if(ik.eq.1)goto 3605
!
! Pattern moves
!
3200 do 3230 i=1,m
3230 x1(i)=2*x2(i)-x1(i)
do 3260 i=1,m
bx(i)=x2(i)
!bvar=var
bmin=min
do 3276 i=1,m
c(i)=x1(i)
call lad_fun(p,c,fv)
min=fv
ik=0
goto 3100
if(fi.eq.1)goto 3615
if(min.ge.bmin)goto 3310
goto 3200
do 3330 i=1,m
x1(i)=bx(i)
!var=bvar
min=bmin
goto 3615
!
End of moves
!

ik=1
goto 3100
if(fi.eq.1)goto 3640
goto 3200
if(fi.eq.1)goto 3640
goto 3600
continue
fi=0
! 3687 d=(.5/sqrt(xn))/(5**iii)

continue
if(iq.ne.0)then
min=bmin
do 4330 i=1,p
   if (i .le. p) AR(i)=bx(i)
   x1(i)=bx(i)
   fv=min
endif
END SUBROUTINE Hooke
END MODULE tree

!***********************************************************************
!***********************************************************************
!***********************************************************************

PROGRAM Bdt2

! this program subset models AR(p)'s for 2-dimensional time series, using
! Burg type recursions, with subset size <=26, km<100, and a max of 10000 obs.

USE tree
INTEGER :: i, wn_known, mc
INTEGER, ALLOCATABLE, DIMENSION (:) :: toplags
TYPE (vector) :: y(10000)
CHARACTER :: h*24
DOUBLE PRECISION :: mean1, mean2, truevf(3)

PRINT*, "%%%%%%%%%%%%%%%%%%%%%% Bivariate SVAR Modeling Program %%%%%%%%%%%%%%%%%"
! read in the time series from a data file
20 write(*,*)
write(*,22)
22 format(5x,'Enter file name of time series for modelling: ',$)
23 h=' ' read(*,*), h
IF (h=='a ') h='sun2.tsm'
open(3,file=h,status='old',err=20)
i=1
25 read(3,*,end=30) y(i)%vec(1), y(i)%vec(2)
i=i+1
if (i.eq.10002) then
write(*,6665)
6665 format(3x,'DATA TRUNCATED AFTER FIRST 10000 OBSERVATIONS.')
i=i-1
goto 30
endif
goto 25
! now mean-correct the obs
mean1=SUM(y%vec(1))/n
mean2=SUM(y%vec(2))/n
ALLOCATE (x(n))
x%vec(1)=(/(y(i)%vec(1), i=1,n)/)
x%vec(2)=(/(y(i)%vec(2), i=1,n)/)
PRINT*, "Do you wish to mean-correct the observations (1=yes, 0=no)?"
READ*, mc
IF (mc==1) THEN
   x%vec(1)=(/(y(i)%vec(1)-mean1, i=1,n)/)
x%vec(2)=(/(y(i)%vec(2)-mean2, i=1,n)/)
END IF
PRINT*, "There are ",n," observations."
WRITE(*,40) x(1), x(n)
40 FORMAT("Firs obs is ",F20.4," last is ",F20.4)

! Enter how many lags will be modeled
PRINT*, "Enter the number of lags to be modeled (<27):
READ*, m
ALLOCATE (toplags(m))

! read in the lags
PRINT*, "Enter the lags:
READ*, (toplags(i), i=1,m)
PRINT*,"The lags are: ", (toplags(i), i=1,m)

! Simulation?
PRINT*, "Is the true white noise covariance matrix known (1=yes, 0=no)?"
READ*, wn_known
IF (wn_known==1) THEN
   PRINT*, "Enter Sigma(1,1), Sigma(2,2), Sigma(1,2):
   READ*, (truevf(i), i=1,3)
ELSE
   ! don't know true WN, so set truevf(1)=0 as flag for routine approx_likelihods
   truevf(1)=0
END IF
! method
PRINT*, "Enter the method for obtaining the reflection coefficients."
PRINT*, "Yule-Walker (1), Burg (2), Vieira-Morf (3), Nuttal-Strand (4):"
READ*, method
SELECT CASE (method)
  CASE (1); stamp="YuWa"
  CASE (2); stamp="Burg"
  CASE (3); stamp="Morf"
  CASE (4); stamp="Nutt"
  CASE DEFAULT
    PRINT*,"Method not in range: ", method
    STOP
END SELECT
CALL make_tree(toplags, truevf)

END PROGRAM Bdt2