Journal of Statistical Software

May 2011, Volume 41, Issue 3. http://www.jstatsoft.org/

State Space Methods in Ox/SsfPack

Matteo M. Pelagatti
Università degli Studi di Milano-Bicocca

Abstract

The use of state space models and their inference is illustrated using the package **SsfPack** for Ox. After a rather long introduction that explains the use of **SsfPack** and many of its functions, four case-studies illustrate the practical implementation of the software to real world problems through short sample programs.

The first case consists in the analysis of the well-known (at least to time series analysis experts) Nile data with a local level model. The other case-studies deal with ARIMA and RegARIMA models applied to the (also well-known) Airline time series, structural time series models applied to the Italian industrial production index and stochastic volatility models applied to the FTSE100 index. In all applications inference on the model (hyper-) parameters is carried out by maximum likelihood, but in one case (stochastic volatility) also an MCMC-based approach is illustrated. Cubic splines are covered in a very short example as well.

Keywords: ARIMA, Kalman filter, state space methods, unobserved components, Gibbs sampling, Ox, **SsfPack**.

1. Introduction

SsfPack (Koopman, Shephard, and Doornik 1999, 2008) is a library of routines for state space modelling and inference written in C and linked to Ox, the efficient matrix language developed by Doornik (2007)[1]. **SsfPack** is written by Siem Jan Koopman, one of the most prolific researchers and developers of state space algorithms, and comes in two different versions. One is free for academic use (the same policy as Ox console) and it is referred to as **SsfPack Basic**, while the other is commercial and its name is **SsfPack Extended**. The latter version includes the functionalities of **SsfPack Basic** plus a set of algorithms that are computationally

[1]At the moment of writing this article the current version of **SsfPack** is 3.0, while that of Ox is 5.1. For **SsfPack** refer to the web page `http://www.ssfpack.com/`, while for Ox the relevant site is `http://www.doornik.com/`

more efficient and allow for exact treatment of the diffuse conditions for the initial state vector. Here only **SsfPack Basic** will be discussed, but the reader may be interested in knowing that for benefitting from the higher computational efficiency of **SsfPack Extended** the state space representation of the model must have orthogonal measurement errors (H_t diagonal) and the disturbances in the transition equations must be uncorrelated with the measurement errors ($\mathsf{E}\eta_t\varepsilon_t^\top = 0$). The book by Koopman *et al.* (2008) contains the complete documentation for both versions of the package, but the reader only interested in **SsfPack Basic** may also refer to Koopman *et al.* (1999).

SsfPack can handle both time-homogeneous and time-varying state space models, and contains various routines for (Kalman) filtering, prediction and smoothing, functions for computing the likelihood, in some cases with analytical scores, and algorithms for generating random sequences from the smoothed distribution of the state vector process. Furthermore, there are functions that facilitate the construction of the system matrices for some classes of frequently used models such as ARIMA, structural time series models (STSM), linear regressions and splines.

The package is usually installed in the folder `package\ssfpack` in the Ox installation directory, and throughout this article it will be assumed so. In order to use the package in an Ox program the line `#include <packages/ssfpack/ssfpack.h>` must be present in the program header[2].

In order to lighten the exposition of the features of **SsfPack**, we introduce slight modifications to the state space notation used in this *Journal of Statistical Software* volume: for $t = 1, \ldots, n$,

$$\begin{aligned} y_t &= c_t + Z_t\alpha_t + \varepsilon_t, \qquad & \varepsilon_t &\sim \mathrm{NID}(0, H_t), \\ \alpha_{t+1} &= d_t + T_t\alpha_t + \eta_t, \qquad & \eta_t &\sim \mathrm{NID}(0, Q_t), \quad C_t = \mathsf{E}\eta_t\varepsilon_t^\top \end{aligned}$$

where y_t, c_t and ε_t are $(p \times 1)$ vectors, α_t, d_t and η_t are $(m \times 1)$ vectors, Z_t is the $(p \times m)$ observation matrix, T_t is the $(m \times m)$ transition matrix, H_t is $(p \times p)$, Q_t is $(m \times m)$ and C_t is $(m \times p)$. With respect to the notation used in this special issue, we added the two system vectors c_t and d_t, imposed the restriction[3] $R = I_m$ and assigned the symbol C_t to the covariance matrix of η_t with ε_t. The system is completed by the initial state specifications

$$\alpha_1 \sim \mathrm{N}(a_1, P_1), \quad \alpha_1 \perp \varepsilon_t, \quad \alpha_1 \perp \eta_t, \qquad \text{for } t = 1, 2, \ldots, n.$$

[2]Those not acquainted with Ox, should take notice of the following:

- `//` denotes a line comment;
- statements end with a semicolon;
- `&` is used for pointers, that is, for passing a variable through its address;
- if `pointer` is a pointer, the pointed variable can be read or assigned using the syntax `pointer[0]`;
- matrix indexing begins with 0;
- `<1,2,3;4,5,6>` defines a matrix constant: `,` switches to the next column and `;` to the next row;
- the operator `~` joints two matrices by columns (side by side), while `|` stacks two matrices;
- arithmetic and logic operators on matrices follow linear algebra conventions (e.g., `*` is matrix product), elementwise operators are preceded by a dot (e.g., `.*` is Hadamard product).

[3]This is actually not a restriction: indeed, if the reader prefers the original form, he/she can redefine the state equation disturbance as $\tilde{\eta}_t = R\eta_t$ and use this in the rest of the paper.

1.1. Formulating the state space system

SsfPack represents the state space system through one vector and three matrices:

$$\delta_t = \begin{bmatrix} d_t \\ c_t \end{bmatrix}, \quad \Phi_t = \begin{bmatrix} T_t \\ Z_t \end{bmatrix}, \quad \Omega_t = \mathsf{E}\left[\begin{bmatrix} \eta_t \\ \varepsilon_t \end{bmatrix} \cdot \begin{bmatrix} \eta_t^\top & \varepsilon_t^\top \end{bmatrix}\right] = \begin{bmatrix} Q_t & C_t \\ C_t^\top & H_t \end{bmatrix}, \quad \Sigma = \begin{bmatrix} P_1 \\ a_1^\top \end{bmatrix}.$$

The matrices Φ_t and Ω_t must always be assigned, while δ_t and Σ are optional. The default assignments for the latter two matrices are

$$\delta_t = 0, \qquad \Sigma = \begin{bmatrix} kI_m \\ 0^\top \end{bmatrix},$$

where k is a very large constant.

In the rest of the article we will refer to the Ox variables containing the system matrices as, respectively, `mDelta`, `mPhi`, `mOmega` and `mSigma`.

In **SsfPack** both homogeneous (i.e., time invariant) and inhomogeneous systems can be implemented. In the first case a state space must be defined through either the pair `mPhi`, `mOmega`, or the triplet `mPhi`, `mOmega`, `mSigma`, or the quartet `mPhi`, `mOmega`, `mSigma`, `mDelta`. Since the user is free to chose any one of these forms, we will use the notation `{Ssf}` to refer to any of the three state space system definitions.

Defining an inhomogeneous system is slightly more complex. For the matrices with time-varying elements additional index matrices of the same dimensions have to be defined. For these matrices, we will use the self-explanatory notation `mJ_Phi`, `mJ_Omega`, `mJ_Delta`. Furthermore, a matrix with the time-varying elements in its rows, say `mXt`, has to be assigned. The elements of the index matrices are all set to -1 except those elements for which the corresponding elements in Φ_t, Ω_t, δ_t are time varying. For those elements that change with time the row numbers of `mXt`, in which the needed time-varying elements are stored, must be indicated in the corresponding position of the relative index matrix.

For example, suppose that only the first element of a (3×2) Φ matrix is time varying, and the time varying coefficient is stored in the third row of `mXt`. Then, the index matrices have to be declared as[4]

```
mJ_Phi = <2, -1; -1, -1; -1, -1>;   mJ_Omega = <>;   mJ_Delta = <>;
```

where `<>` is the empty matrix. As the reader may have noticed from the example, index matrices relative to time-homogeneous system matrices can be set to `<>`.

The four alternative ways to define a state space model in **SsfPack** are summarized below:

```
mPhi, mOmega
mPhi, mOmega, mSigma
mPhi, mOmega, mSigma, mDelta
mPhi, mOmega, mSigma, mDelta, mJ_Phi, mJ_Omega, mJ_Delta, mXt
```

The data matrix, that we will denote as `mYt`, must have the series in rows and the time points in columns. Thus, using the above indexing notation, `mYt` is a $(p \times n)$ matrix. **SsfPack** has no problem to work with data matrices with missing values.

[4]Recall that in Ox arrays and matrices are indexed starting with 0.

1.2. Generating the system matrices for common classes of models

The system matrices of a state space model can be directly assigned by the user, but for some common classes of models there are functions that simplify their assignment.

ARIMA models One of the most used classes of time series models is the ARMA family. If a non-seasonal stationary and invertible ARMA(p,q) model is needed, the function to be used is

```
GetSsfArma(vAr, vMa, dStDev, &mPhi, &mOmega, &mSigma)
```

where `vAr` and `vMa` are vectors containing, respectively, the AR and MA coefficients, `dStDev` is the standard deviation of the white noise processes and the system matrices are passed through their addresses. This function implements the ARMA model in the form

$$\begin{aligned} \alpha_{t+1} &= T\alpha_t + h\xi_t, \qquad \xi_t \sim \text{NID}(0, \sigma_\xi^2), \\ y_t &= Z\alpha_t \end{aligned}$$

with

$$T = \begin{bmatrix} \phi_1 & 1 & 0 & \dots & 0 \\ \phi_2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \phi_{m-1} & 0 & 0 & \dots & 1 \\ \phi_m & 0 & 0 & \dots & 0 \end{bmatrix}, \qquad h = \begin{bmatrix} 1 \\ \theta_1 \\ \vdots \\ \theta_{m-2} \\ \theta_{m-1} \end{bmatrix}, \qquad Z = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix},$$

$m = \max(p, q+1)$, $\phi_i = 0$ for $i > p$ and $\theta_j = 0$ for $j > q$. Thus, $Q = \sigma_\xi^2 hh^\top$ and

$$\Omega = \begin{bmatrix} Q & 0 \\ 0 & 0 \end{bmatrix}.$$

The matrix Σ is built using the unconditional moments of the process α_t.

In addition to the above function, **SsfPack Extended** provides `GetSsfSarima` for seasonal integrated ARMA models (SARIMA). The user of **SsfPack Basic** can design the SARIMA system matrices directly, or take advantage of `GetSsfArma` by opportunely passing the ARMA parameters for a multiplicative seasonal model and modifying the output matrices for integrating the process. As example, the ubiquitous *Airline* model for monthly data will be discussed in Section 3.

Structural time series models Another important class of models both for social and natural sciences is represented by the *structural time series models* (STSM), also referred to as *unobserved components* (UC) time series models. For a short introduction to these models the reader can refer to the introductory article of this special issue. The books by Harvey (1989), West and Harrison (1997), Durbin and Koopman (2001) and Commandeur and Koopman (2007) are all devoted to these kind of models and the interested reader should refer to them for details.

The **SsfPack** function for building the system matrices for a STSM is

```
GetSsfStsm(mStsm, &mPhi, &mOmega, &mSigma)
```

where `mStsm` is a matrix of the form

$$
\begin{aligned}
&\texttt{mStsm} =< \\
&\qquad \begin{array}{llllll}
\texttt{CMP\_LEVEL}, & \sigma_\eta, & 0, & 0, & 0; \\
\texttt{CMP\_SLOPE}, & \sigma_\zeta & 0, & 0, & 0; \\
\texttt{CMP\_SEAS\_DUMMY}, & \sigma_\omega, & s, & 0, & 0; \\
\texttt{CMP\_CYC\_0}, & \sigma_\psi & \lambda_c & \rho & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\texttt{CMP\_CYC\_9}, & \sigma_\psi, & \lambda_c, & \rho, & 0; \\
\texttt{CMP\_IRREG}, & \sigma_\xi, & 0, & 0, & 0
\end{array} \\
&\qquad >;
\end{aligned}
$$

Notice that `CMP_LEVEL`, `CMP_SLOPE`, `CMP_SEAS_DUMMY` etc. are predefined constants, the notation `CMP_SEAS_DUMMY` implies a *stochastic dummy* specification of the seasonality and is alternative to `CMP_SEAS_TRIG`, which defines a (stochastic) trigonometric seasonal component. Up to ten stochastic cycles may be used in the same model. The ordering of the components specification may be different, but the system matrices are built according to the ordering given above (trend, slope, seasonality, cycles, irregular).

The second column of the matrix contains the standard deviations of the components' disturbance with one exception: the standard deviations of the cycle components refer directly to the square root of the cycle variance $\sigma_\psi^2 = \sigma_\kappa^2/(1-\rho^2)$, where σ_κ^2 is the variance of the cycle disturbance.

The third column of the matrix is used only for the seasonal and cyclical components and, in the former case, s is the seasonal periodicity (e.g., 12 for monthly data and 4 for quarterly series), while in the latter case λ_c is the modal frequency of the cycle (e.g., if the modal cycle length is 60 months the corresponding frequency is given by $\lambda_c = 2\pi/60$).

The fourth column is used only in the specification of cycle components and contains the damping factors $\rho \in (0, 1)$.

The fifth column is not used in the **SsfPack Basic** implementation, while in **SsfPack Extended** it is used to specify higher order cycles of the type proposed by Harvey and Trimbur (2003).

The use of `GetSsfStsm` is illustrated with an example in Section 4.

Regression models Regression models can be easily represented in inhomogeneous state space form as

$$\alpha_{t+1} = \alpha_t, \qquad y_t = x_t^\top \alpha_t + \varepsilon_t,$$

where x_t is a vector of regressors at time t and α_t is a vector of regression coefficients. A regression with time-varying coefficients may be obtained by changing the state equations to $\alpha_{t+1} = \alpha_t + \eta_t$. Unless better information is available, the state vector at time $t = 1$ is given a diffuse distribution.

The **SsfPack** function for building the relevant system matrices is

```
GetSsfReg(mXt, &mPhi, &mOmega, &mSigma, &mJ_Phi)
```

where `mXt` is the $(k \times n)$ data matrix. Notice that the function uses only the row dimension of `mXt` and not the content.

Adding regressors to time series models Suppose the matrices `mPhi`, `mOmega` and `mSigma` have been properly assigned for a time-invariant state space model. If regressors have to be added to the model, the function

```
AddSsfReg(mXt, &mPhi, &mOmega, &mSigma, &mJ_Phi)
```

can be used. Again, the function uses only the row dimension, say k, of `mXt` and not its content.

The function call produces

$$\texttt{mPhi} = \begin{bmatrix} I_k & 0 \\ 0 & T \\ 0 & Z \end{bmatrix}, \qquad \texttt{mOmega} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & Q & C \\ 0 & C^\top & H \end{bmatrix}, \qquad \texttt{mSigma} = \begin{bmatrix} -I_k & 0 \\ 0 & P_1 \\ 0 & a_1^\top \end{bmatrix},$$

where the system matrices T, Z, Q, H, P_1 and a_1 are extracted from the passed `mPhi`, `mOmega` and `mSigma`. The returned index matrix `mJ_Phi` is

$$\texttt{mJ\_Phi} = \begin{bmatrix} \underset{(k\times k)}{-1} & \underset{(k\times m)}{-1} \\ \underset{(m\times k)}{-1} & \underset{(m\times m)}{-1} \\ \underset{(1\times k)}{\mathbf{i}} & \underset{(1\times m)}{-1} \end{bmatrix},$$

where -1 denotes a matrix of -1's with the indicated dimensions and $\mathbf{i} = (0, 1, \ldots, k-1)$.

In Section 3 it will be shown how to build a regression model with ARMA errors (generally referred to as *RegARMA* or *ARMAX*) using the functions `GetSsfArma` and `AddSsfReg` jointly.

Nonparametric cubic splines A cubic spline is a function used in interpolation problems and for the approximation of smooth functions observed with noise (a form of nonparametric regression). The interested reader should refer to Green and Silverman (1994) for a general treatment on cubic splines and to Wecker and Ansely (1983) for their relation with signal extraction.

The **SsfPack** function for computing cubic splines is

```
GetSsfSpline(dq, vxt, &mPhi, &mOmega, &mSigma, &mJ_Phi, &mJ_Omega, &mXt)
```

where `dq` is the signal to noise ratio, `vxt` is a (row) vector of abscissa points for which we have noisy observations of the smooth function: $y_i = f(x_i) + \varepsilon_t$.

The approximation to the function can be recovered by passing the system matrices generated by `GetSsfSpline` and the ordinate data y_i to the smoothing functions discussed below. A case-study for this function is not covered in the text of this article, but the interested reader can take a look at the code in `Spline.ox`, in which a (random) continuous curve is generated and sampled at a finite number of points, then some noise is added and a cubic spline is fitted.

1.3. Filtering, smoothing, predicting and forecasting

SsfPack is extremely rich in routines for carrying out predictions, filtering and smoothing. Here, we will just cover those functions useful to the typical "end-user" of state space models.

Recall that in a Gaussian state space system, conditionally on the observations, the state variables are normal and the Kalman filter and smoother are algorithms to compute the conditional moment pairs

$$a_{t|s} = \mathsf{E}[\alpha_t|\mathcal{Y}_s], \qquad P_{t|s} = \mathsf{E}[(\alpha_t - a_{t|s})(\alpha_t - a_{t|s})^\top]$$

with $\mathcal{Y}_s = \{y_1, \ldots, y_s\}$, which are named predictor, filter or smoother when, respectively, $s < t$, $s = t$, $s > t$.

The main function for estimating the first two conditional moments of the state variables is

```
SsfMomentEst(iSel, &mOutput, mYt, {Ssf})
```

where `iSel` is a selection variable whose possible values are `ST_SMO`, `ST_FIL`, `ST_PRED`, `DS_SMO`, for moment smoothing, filtering, prediction and disturbance smoothing, respectively. The matrix `mOutput`, passed to the function by its address, will contain the main output of the function. Naming the first two conditional moments of the *signal* $s_t = c_t + Z_t\alpha_t$ as

$$\theta_{t|s} = \mathsf{E}[s_t|\mathcal{Y}_s], \quad S_{t|s} = \mathsf{E}[(s_t - \theta_{t|s})(s_t - \theta_{t|s})^\top],$$

the matrix `mOutput` will have the following structure:

$$\texttt{mOutput} = \begin{bmatrix} a_{1|\cdot} & \ldots & a_{n|\cdot} \\ \theta_{1|\cdot} & \ldots & \theta_{n|\cdot} \\ \mathrm{diag}\{P_{1|\cdot}\} & \ldots & \mathrm{diag}\{P_{n|\cdot}\} \\ \mathrm{diag}\{S_{1|\cdot}\} & \ldots & \mathrm{diag}\{S_{n|\cdot}\} \end{bmatrix},$$

where the subscript $t|\cdot$ must be substituted either with $t|t-1$, $t|t$ or $t|n$, and the diag operator returns the main diagonal of a square matrix as column vector.

For forecasting (multi-step out-of-sample predictions), the analyst can augment the data matrix with columns of missing values at the end of the observed sample, and then use the function `SsfMomentEst` for either smoothing, filtering or prediction. For example, for forecasting q steps ahead, if the matrix `mYt` contains the observed data, the following line of code could be used[5]: `mYt ~= constant(.NaN, p, q);` where the Ox function `constant` generates a $(p \times q)$ matrix with all elements equal to its first argument, that in this case is the missing value (`.NaN`, not a number).

1.4. Likelihood evaluation

SsfPack offers the following three functions for evaluating the log-likelihood of a state space model:

```
bSuccess = SsfLik(&dLogLik, &dVar, mYt, {Ssf})
bSuccess = SsfLikSco(&dLogLik, &dVar, &mSco, mYt, {Ssf})
bSuccess = SsfLikConc(&dLogLikConc, &dVar, mYt, {Ssf})
```

[5]Notice that if `@` is any binary operator in Ox, then `x @= y` is equivalent to `x = x @ y`.

All functions return 1 if successful and 0 otherwise. The first two functions evaluate the log-likelihood, ℓ, and write its value to the variable `dLogLik` passed through its address. `SsfLikSco` computes also the matrix of scores with respect to the elements of the covariance matrix Ω,

$$S = \frac{\partial \ell}{\partial \Omega},$$

and assign it to `mSco` through its address.

Suppose that ψ_i is an unknown parameter associated only with elements of the matrix Ω, then Koopman and Shephard (1992, equation 3.2) show that the score with respect to ψ_i is given by

$$\frac{\partial \ell}{\partial \psi_i} = \frac{1}{2}\mathrm{tr}\left(S\frac{\partial \Omega}{\partial \psi_i}\right).$$

By all functions the variable `dVar` is assigned the value $\hat{\sigma}^2 = (np-d)^{-1}\sum_{t=1}^{n}\nu_t^\top F_t^{-1}\nu_t$, where d is the number of state variables with diffuse initial conditions, ν_t is the innovation vector and F_t its covariance matrix.

The function `SsfLikConc` computes the profile log-likelihood where a scale parameter is concentrated out, reducing the dimensionality of the estimation problem by one parameter (see Harvey 1989, pp. 126–127). Notice that, while for `SsfLik` and `SsfLikSco` the variable `dVar` will be approximately equal to 1 after maximum likelihood (ML) estimation, for `SsfLikSco` `dVar` contains the scale parameter concentrated out of the likelihood.

In order to make the examples in the next sections easily readable by readers who are not acquainted with Ox, a few lines are now dedicated to the maximization function `MaxBFGS`, upon which we rely for ML estimation of models in state space form. The optimization package is to be imported using `#import <maximize>` in the header of the program. The BFGS maximization function in Ox is

```
iCode = MaxBFGS(func, &vP, &dFunc, 0, bNumDer)
```

where `func` is the name of the objective function, `vP` is the parameter (column) vector (passed through its address) that contains the initial values at input and the value of the parameters at maximum at output, `dFunc` is a variable that will contain the maximum of the objective function at the end of the maximization process, and `bNumDer` is a boolean variable that should be set to 0 if `func` returns analytical derivatives and to 1 otherwise (numerical derivatives will then be computed). The fourth argument of the function is not used in this paper and will always be set to 0. `MaxBFGS` returns a number that can be passed to `MaxConvergenceMsg(iCode)` which returns a human-readable string for assessing the degree of success of the convergence process.

The function `func` is expected by `MaxBFGS` to have the following syntax:

```
bSuccess = func(vP, &dFunc, &vScore, 0)
```

where `vP` is a variable through which the parameter vector is passed, in `dFunc` the function `func` must write the value of the objective function at `vP`, and if the third argument is different from 0, then `func` is expected to write the value of the gradient at `vP` in the variable `vScore`. The function must return 1 in case of success or 0 in case the evaluation failed.

1.5. Simulation

Since the advent of fast and cheap personal computers, simulation has become a standard tool in statistics. **SsfPack** is extremely rich in functions for generating random quantities implied by models in state space form.

For simulating artificial observations from the joint distribution implied by a state space model, the **SsfPack** function is

```
mD = SsfRecursion(mR, {Ssf})
```

where

$$\texttt{mR} = \begin{bmatrix} \eta_0 & \eta_1 & \dots & \eta_n \\ \varepsilon_0 & \varepsilon_1 & \dots & \varepsilon_n \end{bmatrix}$$

is a sequence of random vectors with covariance matrices Ω_t to be generated by the user. For example, in the Gaussian case, i.e., for $(\eta_t^\top \quad \varepsilon_t^\top)^\top \sim \text{NID}(0, \Omega)$, the sequence for $t = 1, \dots, n$ can be generated using the following line of Ox code: `choleski(mOmega)*rann(m+p, n);` Notice that, while ε_0 is irrelevant, since no y_0 vector will be generated, η_0 must have zero mean and identity covariance matrix, since α_1 is generated according to $\alpha_1 = a + L\eta_0$, and the mean vector a and the covariance matrix P of α_1 must be specified in the initial moments matrix `mSigma` as follows:

$$\Sigma = \begin{bmatrix} L \\ a^\top \end{bmatrix},$$

where L satisfies $LL^\top = P$ and can be computed with the function `mL = choleski(mP)`.

The output `mD` contains a pseudo-random realization of the state variables process and of the observable variables process:

$$\text{mD} = \begin{bmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_{n+1} \\ 0 & y_1 & \dots & y_n \end{bmatrix}.$$

Carrying out Bayesian inference using Markov Chain Monte Carlo methods often requires generating sample values from the joint distribution of the state variables conditional on the observed data. If we set $\mathcal{Y}_s = \{y_1, \dots, y_s\}$, **SsfPack** has functions to generate from $\alpha_t|\mathcal{Y}_n$ and $\eta_t|\mathcal{Y}_n$, as well as from $s_t|\mathcal{Y}_n$ and $\varepsilon_t|\mathcal{Y}_n$.

There are two ways to generate samples from the conditional random processes above: one is more efficient, but more involved, the other one is straightforward to implement, but somewhat slower. Here, we cover just the latter method, and refer to the **SsfPack** manual (Koopman *et al.* 2008) for the former.

The function for easily drawing from the smoothed distributions is

```
mD = SsfCondDens(iSel, mYt, {Ssf})
```

`iSel` can be set equal to `ST_SIM` or `DS_SIM` and the function returns a sample from, respectively,

$$\begin{bmatrix} \alpha_1|\mathcal{Y}_n & \dots & \alpha_n|\mathcal{Y}_n \\ s_1|\mathcal{Y}_n & \dots & s_n|\mathcal{Y}_n \end{bmatrix}, \quad \text{or} \quad \begin{bmatrix} \eta_1|\mathcal{Y}_n & \dots & \eta_n|\mathcal{Y}_n \\ \varepsilon_1|\mathcal{Y}_n & \dots & \varepsilon_n|\mathcal{Y}_n \end{bmatrix}.$$

2. Case 1: The local level model applied to the Nile data

In this section it will be shown how the local level model (LLM) can be fit to the volume of the Nile river at Aswan from 1871 to 1970, using maximum likelihood (ML) estimates for the two unknown variances in the model. At the end of the section the model will be extended to allow for a break in the trend.

The **SsfPack** functions treated are `SsfLik`, `SsfLikSco` and `SsfMomentEst`. The complete code can be found in the file `LLM_Nile.ox`, that makes use of the data in `Nile.dat`.

The code is organized in three functions plus the `main()`. Furthermore, since the optimization functions do not allow to pass variables to the objective function other than the parameters with respect to which the optimization is carried out, a global (static) variable is needed: `static decl s_vYt;`

The first function takes the two variances of the LLM (state disturbance first) as arguments and uses them to write the Φ and Ω system matrices to variables passed by address.

```
set_llm(const vP, const amPhi, const amOmega) {
    amPhi[0] = <1;1>;
    amOmega[0] = diag(vP);
}
```

The ready-to-use function `GetSsfStsm` could have been used as well, but in this case we found the direct assignment of the system matrices easier.

The second function is the objective function to be passed to the maximizer and, as noted above, must have a specific structure (see Section 1.4). Before proceeding with the discussion of the function, two things need to be emphasized. Firstly, since the `MaxBFGS` optimizer does not allow constraints on the parameter space, we pass the two variances in their (unconstrained) logarithms, and take the exponents of these log-variances to ensure their non-negativity. This implies that the scores for the log-variances have to be obtained from the scores of the variances. So, if $\ell(\sigma^2)$ is the log-likelihood and $\sigma^2 = \exp(p)$ is the variance, than the score with respect to p is given by

$$\frac{\partial \ell(\sigma^2)}{\partial p} = \sigma^2 \frac{\partial \ell(\sigma^2)}{\partial \sigma^2}, \tag{1}$$

where $\partial \ell(\sigma^2)/\partial \sigma^2$ is computed by `SsfLikSco`. Secondly, for numerical reasons we prefer to return (through the pointer `adLogLik`) the mean log-likelihood. Of course, this choice does not affect the estimates.

Notice that in the function below the variable/pointer `amHessian` is not used, and `avScore` will contain the (mean) score if an address is passed, while if equal to 0 only the (mean) log-likelihood will be computed.

```
loglik(const vLogP, const adLogLik, const avScore, const amHessian)
{
    decl dVar, bSuccess, mPhi, mOmega, mSco;
    decl cT = columns(s_vYt);            // number of observations
    set_llm(exp(vLogP), &mPhi, &mOmega); // builds LLM (see funct. above)
    if (avScore)  // address passed -> analytical scores required
```

```
    {
        bSuccess = SsfLikSco(adLogLik, &dVar, &mSco, s_vYt, mPhi, mOmega);
        adLogLik[0] /= cT;  // mean loglik passed
        avScore[0] = (diagonal(mOmega) .* diagonal(mSco))'/cT; // cf. eq.(1)
        return bSuccess;
    }
    else          // zero passed -> no scores returned
    {
        bSuccess = SsfLik(adLogLik, &dVar, s_vYt, mPhi, mOmega);
        adLogLik[0] /= cT;  // mean loglik passed
        return bSuccess;
    }
}
```

The code is rather self-explanatory and illustrates the use of both `SsfLik` and `SsfLikSco`.

The third function, `acov(const vLogP)` (not reported in the text) computes estimates of the asymptotic covariance matrix of the ML estimators. This function will be used in many other short programs.

The last function is the `main()`, which loads the data, calls the maximization routine, prints the estimates and plots the graphs.

```
main()
{
    s_vYt = loadmat("Nile.dat")';    // loads Nile data and put it in global
    decl dVarEtaInit = 10^3;         // initial guess for Var(eta)
    decl dVarEpsInit = 10^4;         // initial guess for Var(eps)
    decl vLogP = log(dVarEtaInit|dVarEpsInit); // log variances in vector
    decl dLogLik, vScore, mHessian, iRetCode, vP;
    decl mPhi, mOmega, mACov;

    // Estimation
    MaxControl(500, 10, TRUE);   // maxiter=500, report every 10 iter,
                                 // compact output
    iRetCode = MaxBFGS(loglik, &vLogP, &dLogLik, 0, 0); // maxim. of LogLik
    vP = exp(vLogP);             // anti-transform of estimates
    set_llm(vP, &mPhi, &mOmega); // builds the LL model at ML estimates
    mACov = acov(vLogP);         // asymptotic variances for vLogP

    // textual output
    [...]

    // smoothing, prediction, auxiliary residuals
    decl mSmo, mPred, mResid, mForecast;
    SsfMomentEst(ST_SMO,  &mSmo  , s_vYt, mPhi, mOmega);
    SsfMomentEst(ST_PRED, &mPred , s_vYt, mPhi, mOmega);
    SsfMomentEst(DS_SMO,  &mResid, s_vYt, mPhi, mOmega);
    SsfMomentEst(ST_PRED, &mForecast,
```

```
                s_vYt~constant(.NaN,1,10), mPhi, mOmega);

    // plotting graphs
    [...]
}
```

Notice the different features of the function `SsfMomentEst` used for smoothing both the state variables and their disturbances, getting one-step-ahead predictions and forecasts.

Since the optimization has been carried out with respect to log-variances, standard errors for the variances have been obtained using the delta method[6].

By running `LLM_Nile.ox` the following estimates are obtained, while the graphs are shown in Figure 1.

```
Strong convergence using analytical derivatives
Log-likelihood = -632.546
Results for variances using the delta method
                 Estimate    Std.Error
Var(eta)           1469.3       1271.3
Var(eps)           15098.       3139.1
```

As can be noticed by looking at the auxiliary residuals for the trend and relative CI in Figure 1, a break occurred at the end of the XIX century.

Atkinson, Koopman, and Shephard (1997, Section 6.3) suggest a break between 1897 and 1900 as well. The file `LLM_Nile_Break.ox` contains a version of the LLM seen above, where a further trend-break parameter is to be estimated for a date fixed by the user.

We discuss the main modifications made to the previous code. Now there are two global variables:

```
static decl s_vYt;    // data
static decl s_iIndex; // index of the break
```

The function that builds the (now inhomogeneous) system matrices is

```
set_llm(const vP,
        const amPhi, const amOmega, const amSigma, const amDelta,
        const amJ_Phi, const amJ_Omega, const amJ_Delta, const amXt)
{
    amPhi[0]   = <1;1>;          amJ_Phi[0]   = <>;
    amOmega[0] = diag(vP[0:1]);  amJ_Omega[0] = <>;
    amDelta[0] = <0;0>;          amJ_Delta[0] = <0;-1>; // time varying param.
    amSigma[0] = <-1;0>;
    amXt[0] = zeros(1,columns(s_vYt)); // creates dummy variable
    amXt[0][s_iIndex] = vP[2]; // sets impact to vP[2]
}
```

[6] Let $g : \mathbb{R}^k \mapsto \mathbb{R}^m$ be a differentiable function with gradient ∇g, then $\sqrt{n}(\hat{\theta}_n - \theta) \xrightarrow{p} \mathrm{N}(0, \Sigma)$ implies $\sqrt{n}\big(g(\hat{\theta}_n) - g(\theta)\big) \xrightarrow{p} \mathrm{N}\big(0, \nabla g(\theta)^\top \Sigma \nabla g(\theta)\big)$.

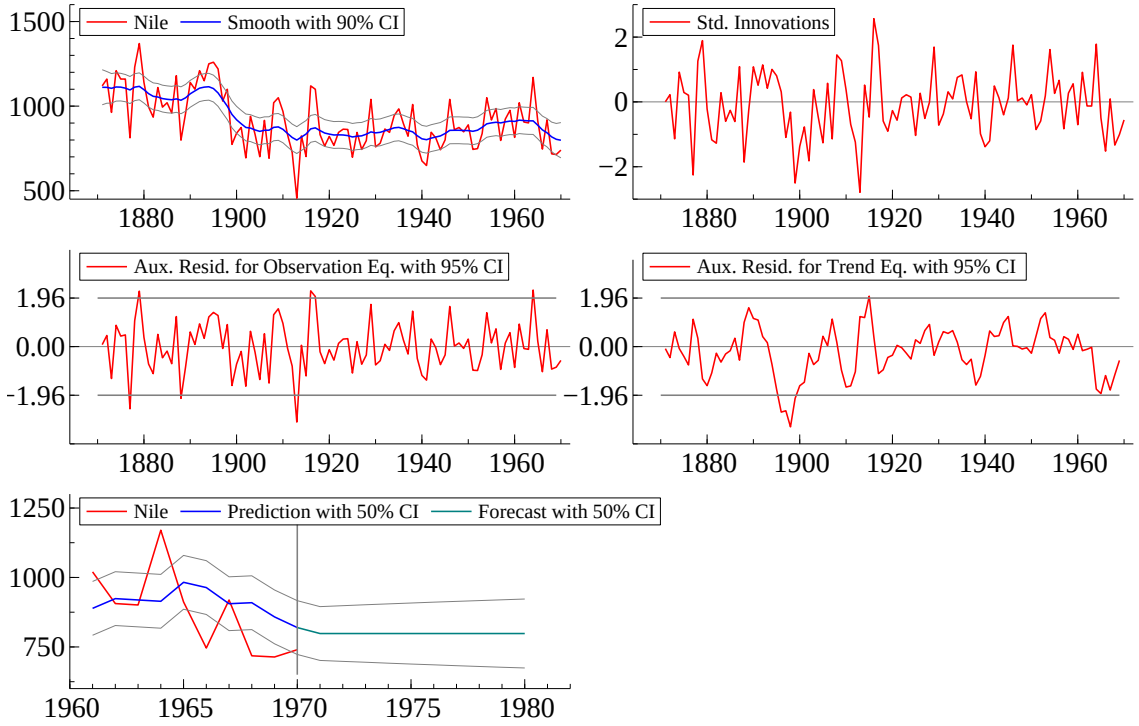

Figure 1: Nile with smoothed trend, standardized prediction errors, auxiliary residuals for the observation equation (for outliers), auxiliary residuals for the trend (for breaks), forecasts.

The other functions (not reported) are not too different, even though in the `loglik` function we show how to compute analytical scores with respect to the variance and numerical scores with respect to the break impact. The textual output for a break in year 1897 is:

```
Strong convergence using analytical derivatives
Log-likelihood = -622.373
                   Estimate    Std.Error
Break                -247.78       28.308
Var(eta)        2.1874e-008  1.4679e-005
Var(eps)              16136.       2292.5
```

Thus, when a break is allowed in the LLM, the trend becomes deterministic since its disturbance variance is virtually zero. The same result was found by Atkinson *et al.* (1997). Figure 2 illustrates this finding with a line plot.

3. Case 2: The airline model applied to the airline time series

State space models are often used for carrying out exact ML estimation of ARIMA models. In this section it will be shown how to cast the airline model (i.e., ARIMA(0,1,1)(0,1,1)$_s$) into state space form, carry out exact ML estimation of the parameters and forecast future

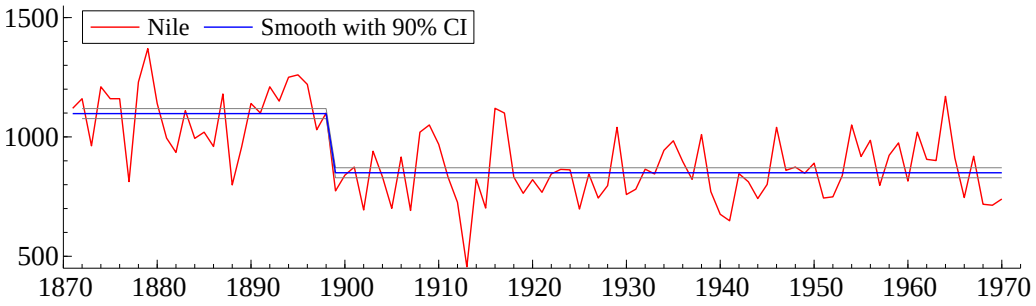

Figure 2: LLM with a break in year 1897 applied to the Nile series.

realizations. Furthermore, the building of regression models with ARMA errors (RegARMA or ARMAX) will be demonstrated.

The **SsfPack** functions treated are `GetSsfArma`, `SsfLikConc`, `SsfMomentEst` and `AddSsfReg`.

The code we are about to illustrate is contained in the file `Airline.ox` and uses the data in `airline.xls`. The global variables needed in the rest of the code are declared as

```
static decl s_vYt;     // data vector
static decl s_cPeriod; // seasonal periodicity
static decl s_dVar;    // innovation variance
```

The first function we report extends the output produced by `GetSsfArma` to generate a general airline model.

```
airline(const vP, const cPeriod, const amPhi, const amOmega, const amSigma)
// vP (3 x 1) vector of MA(1), SMA(1) coefficients and Innovation Std. Dev.
// cPeriod integer value of seasonal periodicity
{
    decl vMA = zeros(cPeriod+1,1); // makes vector of MA coefficients
    vMA[0] = vP[0];                // MA(1) coeff
    vMA[cPeriod-1] = vP[1];        // SMA(1) coeff
    vMA[cPeriod] = vP[0]*vP[1];    // multiplicative "interaction"
    // building the ARMA(0,1)(0,1) part
    decl mPhi, mOmega, mSigma;
    GetSsfArma(<>, vMA, vP[2], &mPhi, &mOmega, &mSigma);
    // building the I(1)(1) part
    decl mA = zeros(cPeriod+1, cPeriod+1);
    decl mB = zeros(cPeriod+1, cPeriod+2);
    mB[cPeriod][0] = 1;
    mA[0][cPeriod-1:] = 1;
    mA[cPeriod][cPeriod] = 1;
    mA[1:cPeriod-1][0:cPeriod-2] = unit(cPeriod-1);
    // putting the pieces together
    amPhi[0] = ( mA~mB )|
               ( zeros(cPeriod+2,cPeriod+1)~mPhi[:cPeriod+1][:cPeriod+1] )|
```

```
                ( 1~zeros(1,2*(cPeriod+1)) );
    amOmega[0] = diagcat(zeros(cPeriod+1,cPeriod+1), mOmega);
    amSigma[0] = diagcat(-unit(cPeriod+1),mSigma);
}
```

This function produces

$$\texttt{mPhi} = \begin{bmatrix} \underset{(s+1)\times(s+1)}{A} & \underset{(s+1)\times(s+2)}{B} \\ \underset{(s+2)\times(s+1)}{0} & \underset{(s+2)\times(s+2)}{T} \\ \underset{1\times(s+1)}{(1,0,\ldots,0)} & \underset{1\times(s+2)}{0} \end{bmatrix}, \quad \texttt{mOmega} = \begin{bmatrix} \underset{(s+1)\times(s+1)}{-I} & \underset{(s+1)\times(s+2)}{0} \\ \underset{(s+2)\times(s+1)}{0} & \underset{(s+2)\times(s+2)}{Q} \end{bmatrix},$$

$$\texttt{mSigma} = \begin{bmatrix} \underset{(s+1)\times(s+1)}{-I} & \underset{(s+1)\times(s+2)}{0} \\ \underset{(s+2)\times(s+1)}{0} & \underset{(s+2)\times(s+2)}{P_1} \\ \underset{1\times(s+1)}{0} & \underset{1\times(s+2)}{a_1^\top} \end{bmatrix},$$

where T, Q, P_1 and a_1 are the state space matrices for the ARMA(0,1)(0,1)$_s$ model produced by `GetSsfArma`, while

$$A = \begin{bmatrix} \underset{1\times(s-1)}{0} & \underset{1\times 2}{1} \\ \underset{(s-1)\times(s-1)}{I} & \underset{1\times 2}{0} \\ \underset{1\times(s-1)}{0} & \underset{1\times 2}{(0,1)} \end{bmatrix}, \quad B = \begin{bmatrix} \underset{(s-1)\times 1}{0} & \underset{(s-1)\times(s-1)}{0} \\ \underset{1\times 1}{1} & \underset{1\times(s-1)}{0} \end{bmatrix}$$

are responsible for the simple and seasonal integration of the ARMA process.

The following function computes the log-likelihood at the parameters' value in `vP`. Notice that only the two MA parameters are to be passed since the innovation variance is concentrated out by the function `SsfLikConc` and assigned to the global variable `s_dVar`. Indeed, the function `airline` is called with arbitrary (unitary) innovation standard deviation.

```
loglik(const vP, const adLogLik, const avScore, const amHessian)
{
    decl cT = columns(s_vYt); // Number of observations
    decl iRetCode, dLogLik, dVar, mPhi, mOmega, mSigma;
    airline(vP|1, s_cPeriod, &mPhi, &mOmega, &mSigma);
    iRetCode = SsfLikConc(adLogLik, &s_dVar, s_vYt, mPhi, mOmega, mSigma);
    adLogLik[0] /= cT; // Mean loglik
    return iRetCode;
}
```

The next function produces forecasts given the model's parameters, the forecast horizon and the desired confidence level in $(0, 1)$. Moreover, if the airline model is built for data in log, the function produces forecasts (under log-normality) and confidence bands for the original data.

```
forecast(const vP, const dStDev, const cHorizon, const dConf, const bAntiLog)
{
    decl mPhi, mOmega, mSigma, mForecast;
    decl cT = columns(s_vYt);           // number of observations
    airline(vP|dStDev, s_cPeriod, &mPhi, &mOmega, &mSigma);
    decl cStDim = columns(mPhi);        // dimension of state vector
    SsfMomentEst(ST_PRED, &mForecast, s_vYt~constant(.NaN,1,cHorizon),
                 mPhi, mOmega, mSigma);
    decl vExpct = mForecast[0][];       // predictions and forecasts
    decl vVar = mForecast[2*cStDim][];  // variance of pred. and forecasts
    decl vStDev = sqrt(vVar);           // Std. dev. of forecasts
    decl vLo, vUp;
    decl dZ = quann(1-(1-dConf)/2);     // quantile for confidence level
    if (bAntiLog) // if data are in log
    {
        vLo = exp(vExpct - dZ*vStDev);  // lower bound for CI
        vUp = exp(vExpct + dZ*vStDev);  // upper bound for CI
        vExpct = exp(vExpct + 0.5*vVar);// expectation of log-normal
    }
    else // if data are not in log
    {
        vLo = vExpct - dZ*vStDev; // lower bound for CI
        vUp = vExpct + dZ*vStDev; // upper bound for CI
    }
    return vExpct|vLo|vUp;
}
```

We omit commenting the function `acov`, which estimates the asymptotic covariance matrix of the ML estimates, and the `main` function, whose role is only the sequential call of the above functions and the nice organization of the results. The estimates obtained by running the code are

```
Maximum likelihood estimation of the Airline model using numerical scores
           Estimate    Std.error      t-ratio      p-value
theta      -0.40183     0.089669      -4.4813  7.4193e-006
Theta      -0.55693     0.073111      -7.6175  2.5757e-014

Innovation Variance: 0.00134827
Mean Log-Likelihood: 1.69921
```

while one year of predictions and two years of forecasts are depicted in Figure 3.

In order to illustrate the use of the ready-to-use functions of **SsfPack** for adding regressors to time series models, we discuss parts of the code in the file `Airline_reg.ox`, which regresses the log-airline time series on a linear time-trend, eleven seasonal sinusoids and leaves the user the option of specifying an ARMA model for the regression errors. The following global variables storing the AR and MA orders and the regressors are added to the previous ones:

```
static decl s_cAR=1;    // AR order, default 1
```

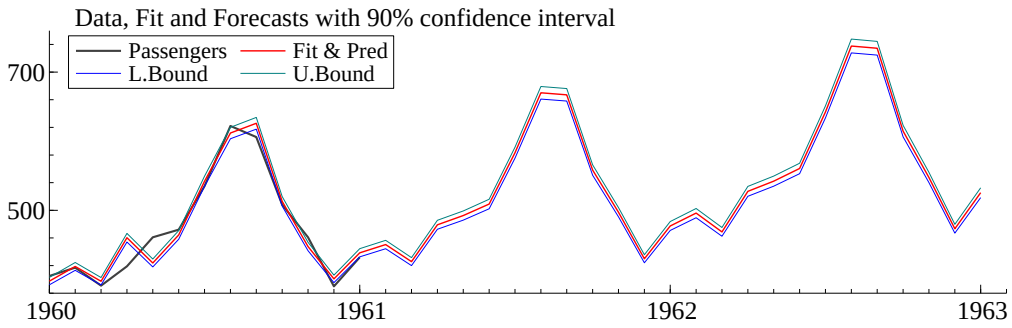

Figure 3: Forecasts for the Airline time series using the ARIMA$(0,1,1)(0,1,1)_s$ model.

```
static decl s_cMA=0;     // MA order, default 0
static decl s_mXt;       // regressors' matrix
```

We just mention that the function `regressors(cPeriod, cT, &mXt)` in the code fills the matrix pointed by `&mXt` (by row) with a constant, a time-counter and `cPeriod`-1 seasonal sinusoids. In `cPeriod` the seasonal periodicity integer must be passed, while the integer in `cT` represents the number of time points to be generated.

The function that computes the log-likelihood value for the ARMA parameters in the vector `vP` is reported below. The function uses the global variables mentioned above to retrieve the orders of the AR and MA parts. Since the regression coefficients are specified as (constant) state variables, they are automatically concentrated out of the likelihood. In our implementation, the sum of the AR and MA orders must be at least one (otherwise no ARMA is specified).

```
loglik(const vP, const adLogLik, const avScore, const amHessian)
{
    decl mPhi, mOmega, mSigma, mDelta=<>;
    decl mJ_Phi=<>, mJ_Omega = <>, mJ_Delta = <>;
    decl iRetCode, vAR = <>, vMA = <>;
    if (s_cMA==0) vAR = vP;
    else if (s_cAR==0) vMA = vP;
    else {
        vAR = vP[0:s_cAR-1];
        vMA = vP[s_cAR:s_cAR+s_cMA-1];
    }
    GetSsfArma(vAR, vMA, 1, &mPhi, &mOmega, &mSigma);
    AddSsfReg(s_mXt, &mPhi, &mOmega, &mSigma, &mJ_Phi);// add regressors
    iRetCode = SsfLikConc(adLogLik, &s_dVar, s_vYt,
                          mPhi, mOmega, mSigma, mDelta,
                          mJ_Phi, mJ_Omega, mJ_Delta, s_mXt);
    adLogLik[0] /= columns(s_vYt); // Mean loglik
    return iRetCode;
}
```

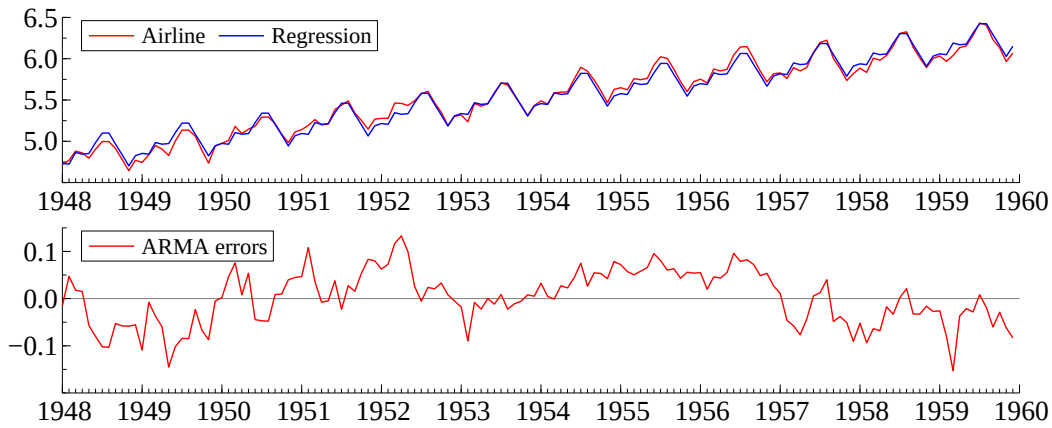

Figure 4: Regression fit and smoothed ARMA errors in the RegARMA model fitted to the airline data.

We do not report the `main` function, but we suggest the reader to go through the code therein to see how the inference on the regression coefficients is obtained through the `SsfMomentEst` function. The code produces Figure 4 and the estimates reported below.

```
AR coefficients:
 Lag   Estimate Std.Error    t-ratio    p-value
   1     0.8736    0.0612    14.2835     0.0000

MA coefficients:
 Lag   Estimate Std.Error    t-ratio    p-value
   1    -0.1802    0.1070    -1.6845     0.0921

Regression coefficients:
        Estimate Std.Error    t-ratio    p-value
Const     4.8128    0.0193  249.0651     0.0000
Trend     0.0100    0.0002   43.6591     0.0000
Cos1     -0.1430    0.0097  -14.7221     0.0000
Cos2     -0.0229    0.0056   -4.0961     0.0000
[...]

Fit statistics:
Innovation Variance: 0.00132045
Mean log-likelihood: 1.5082
```

4. Case 3: A STSM for the Italian industrial production

In this section we show how to fit a complete structural time series model to real data – the Italian industrial production index (IPI) – using the **SsfPack** function `GetSsfStsm`. By complete STSM it is meant that the data generating process is the sum of a *local linear trend*,

a *stochastic trigonometric seasonal*, a *stochastic cycle* and an *irregular* component.

The discussed code is part of the file `STSM_ipi_ita.ox`, that uses data in `IPI_ITA.xls`.

The global variables needed in the code are

```
static decl s_vYt;     // data vector
static decl s_cPeriod; // seasonal periodicity
```

Furthermore, two auxiliary routines that compute the logistic function (`logit`) and its inverse (`antilogit`) are created and used to constrain parameters such as the cycle's damping factor ρ and frequency λ_c. In particular, when a parameter is a variance it is passed in its logarithm from which the non-negative variance can be recovered by taking its exponent, while for parameters constrained to specific intervals we use scaled versions of the logit function, $\text{logit}(x) = [1 + \exp(-x)]^{-1}$, and its inverse, $\text{antilogit}(y) = \log[y/(1 + y)]$.

The function that builds the STSM starting from a vector of parameters is really only a call to `GetSsgStsm`:

```
stsm(const vP, const amPhi, const amOmega, const amSigma)
{   //      component     st.dev.  period   persistence
    decl
    mStsm = CMP_LEVEL~    vP[0]~        0~      0|
            CMP_SLOPE~    vP[1]~        0~      0|
            CMP_CYC_0~    vP[2]~    vP[3]~  vP[4]|
            CMP_SEAS_TRIG~vP[5]~s_cPeriod~      0|
            CMP_IRREG~    vP[6]~        0~      0;
    GetSsfStsm(mStsm, amPhi, amOmega, amSigma);
}
```

The function for computing the (mean) log-likelihood takes unrestricted parameters and maps them into their appropriate subspaces in the way that we just discussed.

```
loglik(const vTransP, const adLogLik, const avScore, const amHessian)
{
    decl vP = exp(vTransP[0:2])|        // positivity constrain
              (logit(vTransP[3])*M_PI)| // 0-pi constrain
              (logit(vTransP[4]))|      // 0-1 constrain
              exp(vTransP[5:6]);        // positivity constrain
    decl cT = columns(s_vYt);
    decl iRetCode, dLogLik, dVar, mPhi, mOmega, mSigma;
    stsm(vP, &mPhi, &mOmega, &mSigma);
    iRetCode = SsfLik(adLogLik, &dVar, s_vYt, mPhi, mOmega, mSigma);
    return iRetCode;
}
```

The `main` function, which we do not report in the text, loads the data, calls the numerical optimizer `MaxBFGS` and produces the graphs in Figure 5 and some simple textual output that we omit. The optimizer achieves only weak convergence (no improvement in line search), indicating a rather flat log-likelihood in a neighborhood of the maximum, but both parameter

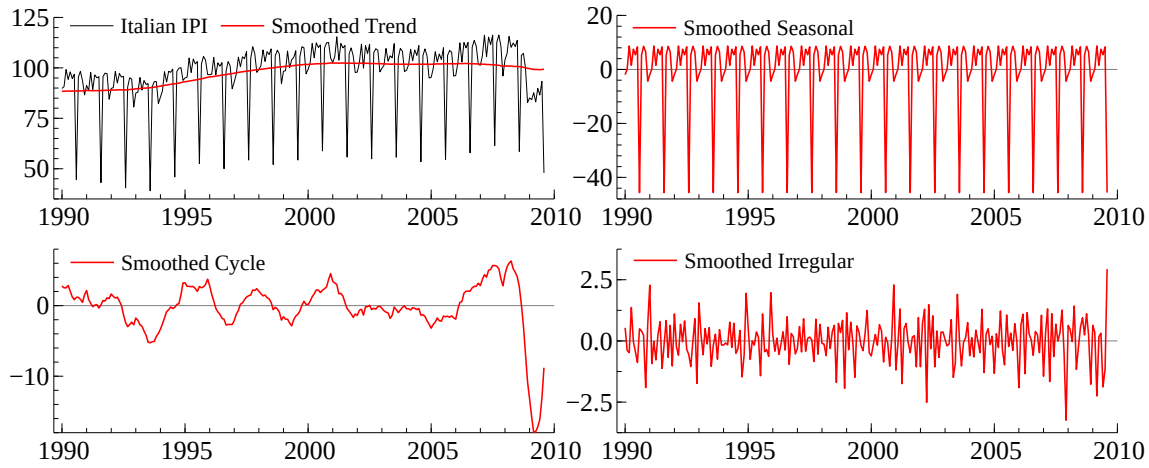

Figure 5: Smoothed components of the Italian industrial production index.

estimates and smoothed components seem quite reasonable and interpretable. For example, the modal frequency of the cycle (0.1678) corresponds to a period of 37 months, which is not surprising for the Italian industrial production.

5. Case 4: The stochastic volatility model

The aim of this last case-study is to show how simple it is for an **SsfPack** user to implement MCMC methods in a non-Gaussian state space framework. The function we focus on is `SsfCondDens` that generates from the state vector distribution conditionally on the observations.

In financial econometrics there are two popular classes of models that deal with time series with time-varying variances: the class of GARCH-type models, where the variance h_t of the process y_t is measurable with respect to the information set $\mathcal{F}_{t-1}$ generated by the past of y_t, and the stochastic volatility (SV, also stochastic variance) models, in which the variance follows a process of its own, which is not measurable with respect to $\mathcal{F}_s$ for any s.

The simplest specification of a SV model is

$$\begin{aligned} y_t &= \exp(h_t/2)\zeta_t \qquad & \zeta_t &\sim \text{NID}(0,1) \\ h_{t+1} &= \nu + \phi h_t + \eta_t \qquad & \eta_t &\sim \text{NID}(0,\sigma_\eta^2), \end{aligned}$$

where the two disturbances are independent. This model is a nonlinear state space form, but can be easily linearized by taking the log of the square of the observable time series:

$$\log y_t^2 = h_t + \log \zeta_t^2.$$

Now the state space is linear, but not Gaussian, since the observation error is a $\log \chi_1^2$ distribution. Harvey, Ruiz, and Shephard (1994) propose to approximate the observation error distribution with a normal with the same first two moments of $\log \chi_1^2$:

$$\log y_t^2 = -1.27 + h_t + \varepsilon_t, \qquad \varepsilon_t \sim \text{NID}(0, \pi^2/2).$$

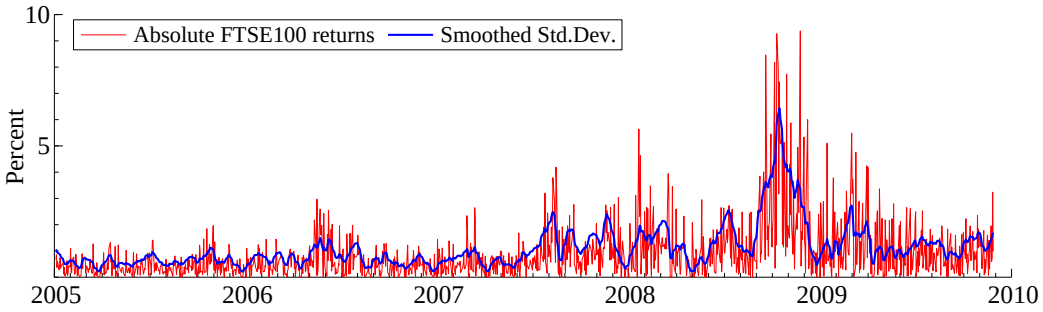

Figure 6: FTSE100 absolute percentage returns and smoothed standard deviation based on pseudo-ML estimation.

The Gaussian likelihood obtained through prediction error decomposition for this model becomes a pseudo-likelihood (also quasi-likelihood) and the Kalman filter and smoother are no more conditional expectations, but only linear projections. Thus, the relative inference provides consistent although inefficient estimators.

The code in `SV_Harvey_ftse.ox`, that makes use of the data in `ftse100.xls`, implements the simple SV model for the daily returns of the FTSE100 index. In order to reduce the dimension of the parameter vector with respect to which the likelihood is maximized, the model has been implemented in the equivalent state space form

$$\log y_t^2 = -1.27 + \mu_t + h_t^* + \varepsilon_t, \qquad \mu_{t+1} = \mu_t, \qquad h_{t+1}^* = \phi h_t^* + \eta_t,$$

where the constant $\mu = \mu_t$ plays the role of the marginal expectation of h_t and $h_t = h_t^* + \mu$. The Ox/SsfPack code for this analysis is very simple and so similar to the previous examples that we do not discuss it in the text. The returned QML estimates are $\mu = -0.023$, $\phi = 0.925$ and $\sigma^2 = 0.289$. Figure 6 plots the FTSE100 absolute percentage returns together with their smoothed standard deviations.

Kim, Shephard, and Chib (1998) consider the same model, but propose a different estimation technique based on Gibbs sampling. Their approach is excellent for demonstrating the simulation facilities of **SsfPack**. The code implementing a slight modification of their Gibbs sampler that we are about to discuss is in the file `SV_Kim_ftse.ox`, and uses the data in `ftse100.xls`.

In Section 3 of their paper, Kim *et al.* (1998) provide a seven-component Gaussian mixture that approximates the $\log \chi_1^2$ distribution very well. In our code the mixture is defined as the (static) global variable `s_mMix`, a (7×3) matrix, in which the first column contains the component probabilities, and the second and the third columns contain, respectively, mean and variance of each component.

The Gibbs sampling recursion for estimating the parameters and the unobservable variance of the SV model consists of the following three steps:

1. given the $\{h\}_{t=1}^n$ sampled in the previous recursion, generate the time series of mixture probabilities $\{w_t\}_{t=1}^n$:

$$\Pr(w_t = i|h_t, y_t) = \frac{\phi\big(\log y^2|h_t + \mu_i, \sigma_i^2\big)\Pr(w_t = i)}{\sum_{i=1}^7 \phi\big(\log y^2|h_t + \mu_i, \sigma_i^2\big)\Pr(w_t = i)}, \qquad i = 1\ldots, 7,$$

where $\phi(x|\mu,\sigma^2)$ denotes the normal density with mean μ and variance σ^2;

2. given the previously sampled $(\nu,\phi,\sigma_\eta^2,\{w_t\})$ generate $\{h_t\}_{t=1}^n$ from the state smoothing density of the state space

$$\begin{aligned}\log y_t^2 &= \mu_{w_t} + h_t + \varepsilon_t, \qquad \varepsilon_t \sim \text{NID}(0,\sigma_{w_t}^2)\\ h_{t+1} &= \nu + \phi h_t + \eta_t, \qquad \eta_t \sim \text{NID}(0,\sigma_t^2)\end{aligned}$$

(this is achieved using the **SsfPack** function `SsfCondDens`);

3. given the previously sampled $\{h_t\}_{t=1}^n$, set $\beta=(\nu,\phi)^\top$,

$$z=\begin{bmatrix}\log y_2^2\\ \vdots\\ \log y_n^2\end{bmatrix}, \quad X=\begin{bmatrix}1 & \log y_1^2\\ \vdots & \vdots\\ 1 & \log y_{n-1}^2\end{bmatrix}, \quad C=(X^\top X)^{-1}, \quad \hat\beta = CX^\top z, \quad v=(z-X\hat\beta)^\top z,$$

and generate first σ_η^2 and than β using their posterior distributions under noninformative priors:

$$\begin{aligned}\sigma_\eta^2|\{y_t\} &\sim \mathcal{IG}((n-1)/2, v(n-1)/2),\\ \beta|\{y_t\},\sigma_t^2 &\sim \mathcal{N}(\hat\beta,\sigma_\eta^2 C),\end{aligned}$$

where $\mathcal{IG}(a,b)$ denotes the Inverse Gamma distribution with parameters a and b.

Before going through the `SV_Kim_ftse.ox` routines that carry out the above steps, it is useful to notice that the following auxiliary functions are therein coded and used:

`phi(mX, mMu, mSigma)` returns a matrix containing the values of the Gaussian densities of the points in `mX` computed for mean and standard deviations, respectively, in `mMu` and `mSigma`;

`randiscrete(mProbs)` returns a $(1\times n)$ vector of random numbers chosen from $0,1,\ldots,k-1$, according to the probabilities specified in the $(k\times n)$ matrix `mProbs` (every column sums to 1);

`ranmvn(vM, mS, c)` returns a matrix $(k\times c)$ of `c` random vectors generated from a normal distribution with mean $(k\times 1)$ vector `vM` and covariance $(k\times k)$ matrix `mS`.

The function that accomplishes step 1 of the Gibbs sampler is `w_gen(const vH, const vLogY2)` that generates the indices of the mixture given the variances (`vH`) and the transfomed observations (`vLogY2`).

The function for step 2 is

```
h_gen(const vArPars, const vW, const vLogY2)
{
    // parameters' extraction
    decl dNu = vArPars[0], dPhi = vArPars[1], dSig2 = vArPars[2];
    // builds time-varying system matrices
    decl mPhi = dPhi|1, mJ_Phi = <>;
```

```
    decl mDelta = dNu|0, mJ_Delta = (-1)|0;
    decl mOmega = diag(dSig2|0), mJ_Omega = <-1,-1;-1,1>;
    decl mSigma = (fabs(dPhi)<1)?               // AR(1) stationary?
            (dSig2/(1-dPhi^2))|(dNu/(1-dPhi)) : // yes
            (-1)|0;                             // no
    decl mXt = (s_mMix[vW][1]') | (s_mMix[vW][2]');
    // generates from the smoothing distribution
    decl mD = SsfCondDens(ST_SIM, vLogY2,
                          mPhi, mOmega, mSigma, mDelta,
                          mJ_Phi, mJ_Omega, mJ_Delta, mXt);
    return mD[0][];
}
```

This step is really made easy by **SsfPack** as the function `SsfCondDens` does all the work very efficiently.

Step 3 is carried out by `arpars_gen(const vH)` that generates the autoregression parameters given the log-variances in `vH`.

The Gibbs sampler iterations are implemented in the function `sv_sim(const vLogY2, const cN, const amH, const amArPars, const amW)`, where the transformed data are passed through `vLogY2`, the number of simulations is specified in `cN` and the other three parameters are pointers to variables that will host the simulated log-variance time series, the AR parameters and the mixture weights time series, respectively.

In principle, the initial values for the Gibbs sampler could be arbitrary, since, eventually, the ergodic Markov chain will start exploring the sample space with the right marginal probabilities, but initial values in a probability-dense neighbor of the sample space will speed up this convergence. In the function `sv_sim` we chose to fix the initial $\{h_t\}$ through some kind of exponential smoothing of $\log y_t^2$ and then generate the AR parameters and weights using the steps 3 and 1 of the Gibbs sampler. Generally a burn-in sample of simulations is discarded in order to make the Markov chain "forget" the initial values.

The `main` function loads the data, calls the Gibbs sampler (`sv_sim`) and, after 21000 iterations, the first 1000 of which are discarded, produces the plot in Figure 7 and the following textual output.

```
            Mean    Median      2.5%     97.5%
Nu        0.0002    0.0002   -0.0100    0.0097
Phi       0.9888    0.9892    0.9767    0.9980
Sig2      0.0305    0.0287    0.0160    0.0559
```

The smoothed variance estimates in Figure 7 are obtained as averages of the 20000 Gibbs samples for $\exp\{h_t/2\}$. By taking sample quantiles of the same quantities, confidence intervals (or credible intervals if we take a Bayesian approach) can be computed. The main difference between these results and our pseudo-ML estimates is the higher persistence of the log-variance process h_t. This can be seen both from the larger value of the AR(1) coefficient ϕ and in the smoothed variance graphs.

In order to assess if the Gibbs sample is large enough to sufficiently explore the parameter space the code in `SV_Kim_ftse.ox` produces also autocorrelation functions and kernel density estimates of the MCMC samples (not reported here).

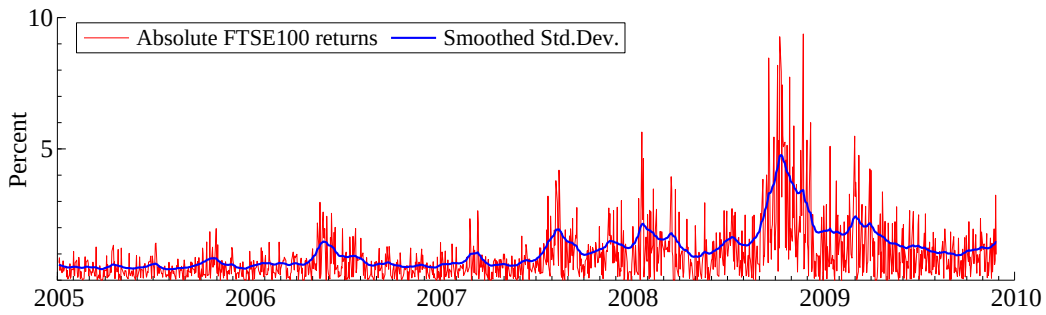

Figure 7: FTSE100 absolute percentage returns and smoothed standard deviation based on MCMC estimation.

6. Conclusions

In this tour trough the Ox package **SsfPack**, we visited just the higher-level functions which are those that the typical user will utilize. Advanced users will find a host of other functions for simulation and filtering/smoothing purposes that can make these operations even more efficient and accurate (see Durbin and Koopman 2001; Koopman *et al.* 2008).

In particular, the commercial version of the software, **SsfPack Extended**, implements exact algorithms to deal with diffuse initial conditions for the state variables and efficient routines to carry out computations in systems with large matrices, in some cases by exploiting the fact that these are typically sparse (i.e., contain many zeros).

Furthermore, **SsfPack Extended** works under Windows 32-bit, Windows 64-bit, OS X, Linux 32-bit and Linux 64-bit, while **SsfPack Basic** is only for Windows 32-bit.

If compared to other software for state space modelling we are acquainted with, **SsfPack** is faster and wider-ranging. On the other hand, something that is missing in **SsfPack**, but present in software written with engineering in mind, such as MATLAB, is some implementation of the Extended Kalman Filter (EKF), that is, the possibility of letting the system matrices depend on predicted state variables.

Any researcher that makes extensive use of models in state space form can only benefit from trying **SsfPack**, and after some use he/she will probably find it irreplaceable for the speed and stability of its algorithms.

Information, updates and other example programs can be found at the official **SsfPack** Internet site `http://www.ssfpack.com/`, which is maintained by S. J. Koopman, the author of **SsfPack**.

References

Atkinson AC, Koopman SJ, Shephard N (1997). "Detecting Shocks: Outliers and Breaks in Time Series." *Journal of Econometrics*, (80), 387–422.

Commandeur JJF, Koopman SJ (2007). *An Introduction to State Space Time Series Analysis*. Oxford University Press, Oxford.

Doornik JA (2007). *Object-Oriented Matrix Programming Using Ox*. 3rd edition. Timberlake Consultants Press, London.

Durbin J, Koopman SJ (2001). *Time Series Analysis by State Space Methods*. Number 24 in Oxford Statistical Science Series. Oxford University Press, Oxford.

Green P, Silverman BW (1994). *Nonparametric Regression and Generalized Linear Models: A Roughness Penalty Approach*. Chapman & Hall, London.

Harvey AC (1989). *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, Cambridge.

Harvey AC, Ruiz E, Shephard N (1994). "Multivariate Stochastic Variance Models." *Review of Economic Studies*, **61**(2), 247–264.

Harvey AC, Trimbur T (2003). "General Model-Based Filters for Extracting Cycles and Trends in Economic Time Series." *Review of Economics and Statistics*, **85**(2), 244–255.

Kim S, Shephard N, Chib S (1998). "Stochastic Volatility: Likelihood Inference and Comparisons with ARCH models." *Review of Economic Studies*, **65**(3).

Koopman SJ, Shephard N (1992). "Exact Score for Time Series Models in State Space Form." *Biometrika*, **79**(4), 823–826.

Koopman SJ, Shephard N, Doornik JA (1999). "Statistical Algorithms for Models in State Space Using **SsfPack** 2.2." *Econometrics Journal*, **2**(1), 113–166.

Koopman SJ, Shephard N, Doornik JA (2008). ***SsfPack** 3.0: Statistical Algorithms for Models in State Space Form*. Timberlake Consultants Press, London.

Wecker WE, Ansely CF (1983). "The Signal Extraction Approach to Nonlinear Regression and Spline Smoothing." *Journal of the American Statistical Association*, **78**(381), 81–89.

West M, Harrison J (1997). *Bayesian Forecasting and Dynamic Models*. 2nd edition. Springer-Verlag, New York.

Affiliation:

Matteo M. Pelagatti
Department of Statistics
Università degli Studi di Milano-Bicocca
Via Bicocca degli Arcimboldi, 8
I-20126 Milano, Italy
E-mail: matteo.pelagatti@unimib.it
URL: http://www.statistica.unimib.it/utenti/p_matteo/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 41, Issue 3 *Submitted:* 2010-01-15
May 2011 *Accepted:* 2010-11-11